Bilkent University

Department of Computer Engineering

CS 353 Database Systems

Project Design Report

Online Course Platform

Group 1

Abdullah Can Alpay 21702686 - Section 2

Cemre Biltekin - 21802369 - Section 1

Mustafa Yaşar - 21702808 - Section 1

Oğuz Kaan İmamoğlu - 21702233 - Section 1

Instructor: Uğur Güdükbay

Teaching Assistant: Mustafa Can Çavdar

Revised E/R Model

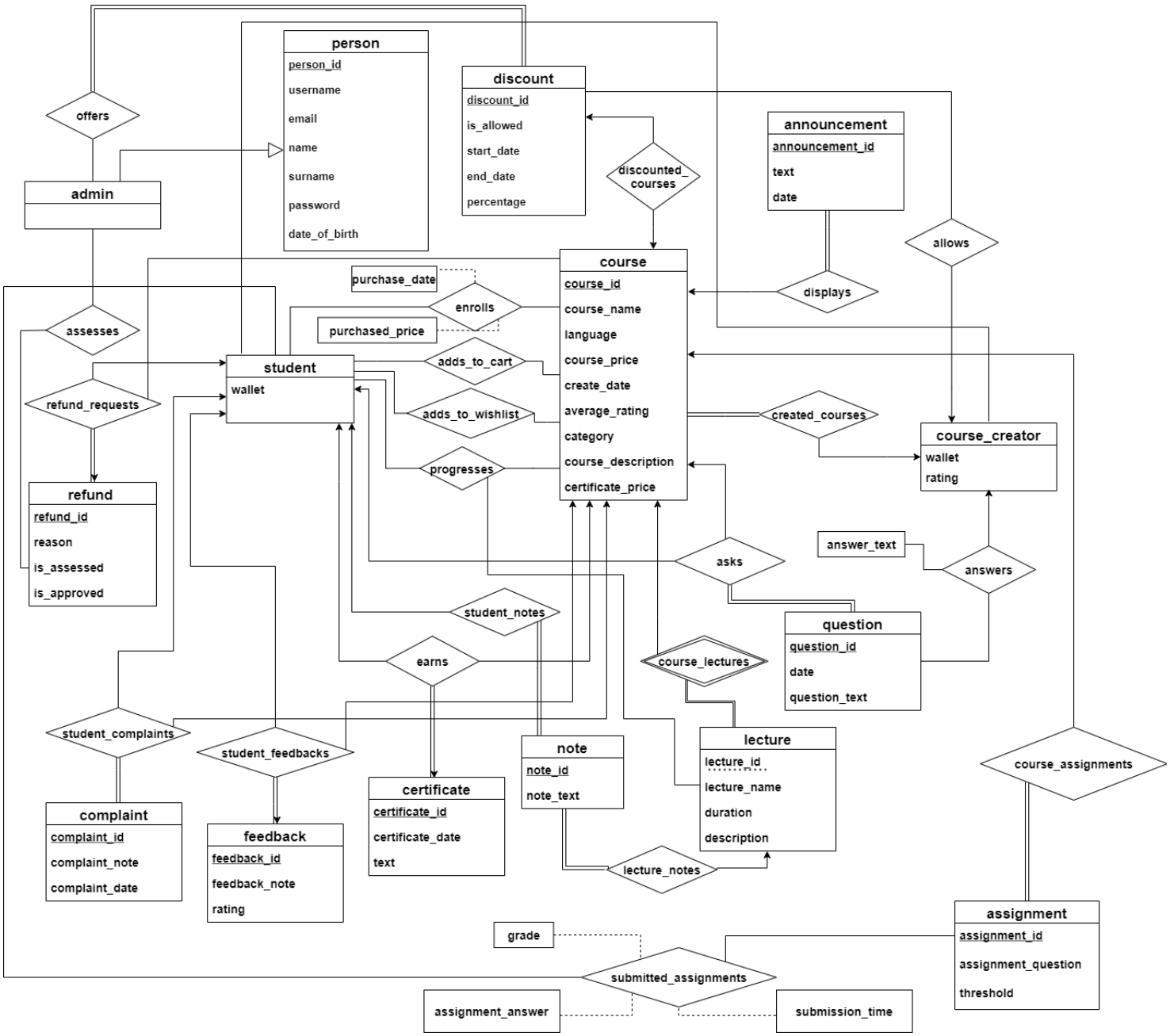

Fig.1: Entity Relationship (E/R) Diagram of Online Course Platform System

Changes in Revised E/R Model

Previously, the E/R diagram was faulty because a weak entity relationship between weak entities (*lecture* and *note* were both weak entities, *note* being a weak entity dependent on *lecture*) was constructed. *Note* entity also had a weak entity relationship with *student* which meant *note* weak entity had two entities to be a weak entity of (weak entity related to both *student* and *lecture*), which is not possible. Similarly, *lecture* also had two weak relationships (*lecture_notes* and *course_lectures*). We eliminated this weak entity problem by changing the *note* entity to a strong entity.

A new relationship *discounted_courses* between discount and course is created with appropriate cardinality and participation constraints. Thus, the foreign key attribute written in the *discount* entity is deleted on the diagram as a fix.

In the previous E/R diagram total participation constraints were missing for some relationships. Thus, we updated the following relations into total participation relation:

- *discount* total participates in the *offers* relationship.
- *certificate* total participates in the *earns* ternary relationship.
- *note* total participates in *student_notes* and *lecture_notes* relationships.
- *refund* total participates in *refund_requests* relationship.
- *question* total participates in *asks* relationship.
- *assignment* total participates in *course_assignments* relationship.
- *complaint* total participates in *student_complaints* relationship.

Previously, *course* entity did not participate in the *student_complaints* ternary relationship. It is fixed by making the *course* participate in the *student_complaints* relationship. Thus, the faulty written foreign key (course_id) is eliminated from the E/R diagram and is represented with this relationship.

Ternary relationship *created_courses* of *course*, *course_creator* and *announcement* could be turned into binary relationships in the E/R diagram for clarity. To fix this, we have splitted *created_course* into two relationships like the following:

- *announcement* and *course* has the *displays* relationship.
- *course* and *course_creator* has *created_course* relationship.

Cardinality of the following ternary relationships are fixed:

- *student_feedback* ternary relationship is updated.
- *asks* ternary relationship is updated.
- *refund_requests* ternary relationship is updated.
- created_courses ternary relationship is deleted and turned into two binary relationships as stated before.

New attribute “surname” is added to the *person* entity. Since premium feature is deleted, related attributes and entities are also eliminated. *premium_student* entity is deleted.

premium_course attribute from the *course* entity is deleted. Instead of a premium feature, assignment as an extra feature is enhanced further. The deadline attribute from the *assignments* entity is removed. Furthermore, in *submitted_assignments* we add two additional attributes which are grades and submission_time. When the user completes an assignment, the course creator will announce the student's grade. Additionally, is_completed attribute is redundant, we have deleted that attribute from the *progresses* relationship.

New relationship attribute “purchased_price” is added to the enrolls relationship so that the price the student has paid for the course is saved. is_approved attribute is deleted from the feedback entity. Additionally, two new entities are added (is_approved and is_assessed) to the refund entity.

Relational Model

person

Relational Schema: person(person_id, username, email, name, surname, password, date_of_birth)

Attribute Domains:

- person_id: int
- username: varchar(24)
- email: varchar(64)
- name: varchar(32)
- surname: varchar(32)
- password: varchar(32)
- date_of_birth: date

Candidate Keys: {person_id}, {username}, {email}
Primary Key: {person_id}
Foreign Key: None

Functional Dependencies:

- person_id -> username, email, name, surname, password, date_of_birth
- username -> person_id, email, name, surname, password, date_of_birth
- email -> person_id, username, name, surname, password, date_of_birth

Normal Form: BCNF. Because for each non-trivial functional dependency, the determinant is a superkey.

Table Definition:

```
CREATE TABLE person(
        person_id INT PRIMARY KEY AUTO_INCREMENT,
```

```
        username VARCHAR(24) NOT NULL UNIQUE,
        email VARCHAR(64) NOT NULL UNIQUE,
        name VARCHAR(32) NOT NULL,
        password VARCHAR(32) NOT NULL,
        date_of_birth DATE DEFAULT NULL
    ) ENGINE = INNODB;
```

student

Relational Schema: student(<u>student_id</u>, wallet)

Attribute Domains:

- student_id: int
- wallet: numeric(10, 2)

Candidate Key: {student_id}
Primary Key: {student_id}
Foreign Key: student_id references person(person_id)

Functional Dependencies:

- student_id -> wallet

Normal Form: BCNF. Because student_id is a superkey.

Table Definition:

```
CREATE TABLE student(
        student_id INT PRIMARY KEY,
        FOREIGN KEY (student_id) REFERENCES person(person_id)
        ON DELETE CASCADE,
        ON UPDATE RESTRICT,
        wallet NUMERIC(10, 2) DEFAULT 0,
        CONSTRAINT check_wallet
                CHECK (wallet >= 0)
) ENGINE = INNODB;
```

course_creator

Relational Schema: course_creator(<u>course_creator_id</u>, wallet, rating)

Attribute Domains:

- course_creator_id: int
- wallet: numeric(10, 2)
- rating: numeric (2, 1)

Candidate Key: {course_creator_id}
Primary Key: {course_creator_id}
Foreign Key: course_creator_id references person(person_id)

Functional Dependencies:

- course_creator_id -> wallet, rating

Normal Form: BCNF. Because course_creator_id is a super key.

Table Definition:

```
CREATE TABLE course_creator(
        course_creator_id INT,
        FOREIGN KEY (course_creator_id) REFERENCES person(person_id)
        ON DELETE CASCADE
        ON UPDATE RESTRICT,
        wallet NUMERIC(10, 2) DEFAULT 0,
        RATING NUMERIC(2,1) DEFAULT 0,
        PRIMARY KEY (course_creator_id),
        CONSTRAINT check_course_creator_rating_positivity
                CHECK (rating >= 0 AND rating <= 5),
        CONSTRAINT check_course_creator_wallet_positivity
                CHECK (wallet >= 0)
) ENGINE = INNODB;
```

course

Relational Schema: course(course_id, course_name, language,
course_price, create_date, average_rating,
category, course_description,
certificate_price, course_creator_id)

Attribute Domains:

- course_id: int
- course_name: varchar(60)
- language: varchar(50)
- course_price: numeric(5, 2)
- create_date: date
- average_rating: numeric(2, 1)
- category: varchar(50)
- course_description: varchar(200)
- certificate_price: numeric(5, 2)
- course_creator_id: int

Candidate Key: {course_id}
Primary Key: {course_id}
Foreign Key: course_creator_id references course_creator(course_creator_id)

Functional Dependencies:

- course_id -> course_name, language, course_price, create_date, average_rating, category, course_description, certificate_price

Normal Form: BCNF. Because course_id is a superkey.

Table Definition:

```
CREATE TABLE course(
        course_id INT PRIMARY KEY AUTO_INCREMENT,
        course_name VARCHAR(60) NOT NULL,
        language VARCHAR(50),
        course_price NUMERIC(5, 2) DEFAULT 0,
        create_date DATE,
        average_rating NUMERIC(2, 1) DEFAULT 0,
        category VARCHAR(30) CHECK (category in
                ('Web Development', 'Mobile Software Development',
                'Programming Languages', 'Game Development', 'Database Management System', 'Business', 'Management', 'Economics', 'Finance', 'Information Technology', 'Cyber Security', 'Maths', 'Gastronomy', 'Others')),
        course_description VARCHAR(200),
        certificate_price NUMERIC(5, 2) DEFAULT 0,
        course_creator_id INT,
        FOREIGN KEY (course_creator_id) REFERENCES
        course_creator(course_creator_id)
        ON DELETE CASCADE
        ON UPDATE RESTRICT
        CONSTRAINT check_course_price_validity CHECK (course_price >= 0),
        CONSTRAINT check_average_rating_validity CHECK (average_rating >= 0 AND average_rating <= 5),
        CONSTRAINT check_certificate_price_validity CHECK (certificate_price >= 0),
        )
) ENGINE = INNODB;
```

enrolls

Relational Schema: enrolls(<u>student_id</u>, <u>course_id</u>, purchased_price, purchase_date)

Attribute Domains:
- student_id: int
- course_id: int
- purchased_price: numeric(5, 2)
- purchase_date: date

Candidate Key: {student_id, course_id}
Primary Key: {student_id, course_id}
Foreign Keys:
- student_id references student(student_id)
- course_id references course(course_id)

Functional Dependencies:
- student_id, course_id -> purchased_price, purchase_date

Normal Form: BCNF. Because { student_id, course_id } is a superkey.

Table Definition:

```
CREATE TABLE enrolls(
        student_id INT,
        course_id INT,
        purchased_price NUMERIC(5, 2),
        purchase_date DATE,
        PRIMARY KEY(student_id, course_id),
        FOREIGN KEY (student_id) REFERENCES student(student_id)
        ON DELETE CASCADE
        ON UPDATE RESTRICT,
        FOREIGN KEY (course_id) REFERENCES course(course_id)
        ON DELETE CASCADE
        ON UPDATE RESTRICT,
) ENGINE = INNODB;
```

adds_to_cart

Relational Schema: adds_to_cart(<u>student_id, course_id</u>)

Attribute Domains:
- student_id: int
- course_id: int

Candidate Key: {student_id, course_id}
Primary Key: {student_id, course_id}
Foreign Keys:

- student_id references student(student_id)
- course_id references course(course_id)

Functional Dependencies: No non-trivial functional dependency.

Normal Form: BCNF. Because there is no non-trivial functional dependency.

Table Definition:

```
CREATE TABLE adds_to_cart(
        student_id INT,
        course_id INT,
        PRIMARY KEY(student_id, course_id),
        FOREIGN KEY (student_id) REFERENCES student(student_id)
        ON DELETE CASCADE
        ON UPDATE RESTRICT,
        FOREIGN KEY (course_id) REFERENCES course(course_id)
        ON DELETE CASCADE
        ON UPDATE RESTRICT
) ENGINE = INNODB;
```

adds_to_wishlist

Relational Schema: adds_to_wishlist(<u>student_id, course_id</u>)

Attribute Domains:

- student_id: int
- course_id: int

Candidate Key: {student_id, course_id}
Primary Key: {student_id, course_id}
Foreign Keys:

- student_id references student(student_id)
- course_id references course(course_id)

Functional Dependencies: No non trivial functional dependency.

Normal Form: BCNF. Because there is no non-trivial functional dependency.

Table Definition:

```
CREATE TABLE adds_to_wishlist(
        student_id INT,
        course_id INT,
        PRIMARY KEY(student_id, course_id),
        FOREIGN KEY (student_id) REFERENCES student(student_id)
        ON DELETE CASCADE
        ON UPDATE RESTRICT,
        FOREIGN KEY (course_id) REFERENCES course(course_id)
        ON DELETE CASCADE
        ON UPDATE RESTRICT,
) ENGINE = INNODB;
```

progresses

Relational Schema: progresses(<u>student_id</u>, <u>course_id</u>, <u>lecture_id</u>)

Attribute Domains:

- student_id: int
- course_id: int
- lecture_id: int

Candidate Key: {student_id, course_id, lecture_id}
Primary Key: {student_id, course_id, lecture_id}
Foreign Keys:

- student_id references student(student_id)
- {course_id, lecture_id} references lecture(lecture_id)

Functional Dependencies: No non-trivial dependency.

Normal Form: BCNF. Because there is no non-trivial dependency.

Table Definition:

```
CREATE TABLE progresses(
        student_id INT,
        course_id INT,
        lecture_id INT,
        PRIMARY KEY(student_id, course_id, lecture_id),
        FOREIGN KEY (student_id) REFERENCES student(student_id)
        ON DELETE CASCADE
        ON UPDATE RESTRICT,
        FOREIGN KEY (course_id, lecture_id) REFERENCES lecture(course_id, lecture_id)
        ON DELETE CASCADE
        ON UPDATE RESTRICT,
) ENGINE = INNODB;
```

admin

Relational Schema: admin(admin_id)

Attribute Domains:
- admin_id: int

Candidate Key: {admin_id}
Primary Key: {admin_id}
Foreign Key: admin_id references person(person_id)

Functional Dependencies: No non-trivial functional dependency.

Normal Form: BCNF. Because there is no non-trivial functional dependency.

Table Definition:

```
CREATE TABLE admin(
        admin_id INT PRIMARY KEY,
        FOREIGN KEY admin_id REFERENCES person(person_id)
        ON DELETE CASCADE,
        ON UPDATE RESTRICT
) ENGINE = INNODB;
```

lecture

Relational Schema: lecture(course_id, lecture_id, lecture_name, duration, description)

Attribute Domains:
- course_id: int
- lecture_id: int
- lecture_name: varchar(64)
- duration: time
- description: varchar(360)

Candidate Key: {course_id, lecture_id}
Primary Key: {course_id, lecture_id}
Foreign Key: course_id references course(course_id)

Functional Dependencies:
- lecture_id -> lecture_name, duration

Normal Form: BCNF. Because lecture_id is a superkey.

Table Definition:

```
CREATE TABLE lecture(
        course_id INT,
        lecture_id INT,
        lecture_name VARCHAR(64) NOT NULL,
        duration TIME NOT NULL,
        description VARCHAR(360),
        PRIMARY KEY (course_id, lecture_id),
        FOREIGN KEY (course_id) REFERENCES course(course_id)
        ON DELETE CASCADE,
        CONSTRAINT check_duration_validity
                CHECK(duration > 0)
) ENGINE = INNODB;
```

assignment

Relational Schema: assignment(<u>assignment_id</u>, assignment_question, threshold, course_id)

Attribute Domains:

- assignment_id : int
- assignment_question: varchar(100)
- threshold: int
- course_id: int

Candidate Key: {assignment_id}
Primary Key: {assignment_id}
Foreign Key: course_id references course(course_id)

Functional Dependencies:

- assignment_id -> assignment_question

Normal Form: BCNF. Because assignment_id is a superkey.

Table Definition:

```
CREATE TABLE assignment(
                assignment_id INT PRIMARY KEY,
                assignment_question VARCHAR(100) NOT NULL,
                assignment_threshold INT,
                course_id INT,
                FOREIGN KEY (course_id) REFERENCES course(course_id)
                ON DELETE CASCADE
                ON UPDATE RESTRICT,
```

```
        CONSTRAINT check_threshold
                CHECK (threshold>0)
    ) ENGINE = INNODB;
```

submitted_assignments

Relational Schema: submitted_assignments(assignment_id, student_id, submission_time, assignment_answer, grade)

Attribute Domains:

- assignment_id : int
- student_id: int
- submission_time: timestamp
- assignment_answer: varchar(300)
- grade: int

Candidate Key: {assignment_id, student_id, submission_time}
Primary Key: {assignment_id, student_id, submission_time}
Foreign Keys:

- assignment_id references assignment(assignment_id)
- student_id references student(student_id)

Functional Dependencies:

- assignment_id, student_id, submission_time -> assignment_answer, grade

Normal Form: BCNF. Because {assignment_id, student_id, submission_time} is a superkey.

```
CREATE TABLE  submitted_assignment(
        assignment_id INT,
        student_id INT,
        submission_time TIMESTAMP,
        assignment_answer VARCHAR(300) NOT NULL,
        grade INT,
        PRIMARY KEY (assignmet_id, student_id, submission_time),
        FOREIGN KEY (assignment_id) REFERENCES assignment(assignment_id)
        ON DELETE CASCADE
        ON UPDATE RESTRICT,
        FOREIGN KEY (student_id) REFERENCES student(student_id)
        ON DELETE CASCADE
        ON UPDATE RESTRICT,
        CONSTRAINT check_grade_validity
                CHECK(grade > 0)
    ) ENGINE = INNODB;
```

question

Relational Schema: question(question_id, date, question_text)

Attribute Domains:
- question_id: int
- date. date
- question _text: varchar(200)

Candidate Keys: {question_id}
Primary Keys: {question_id}
Foreign Keys: None

Functional Dependencies:
- question_id -> date, question_text

Normal Form: BCNF. Because question_id is a superkey.

```
CREATE TABLE question(
                question_id INT PRIMARY KEY,
                question_text VARCHAR(200) NOT NULL,
                date DATE NOT NULL
        ) ENGINE = INNODB;
```

announcement

Relational Schema: announcement(announcement_id, text, date, course_id)

Attribute Domains:
- announcement_id: int
- text: varchar(500)
- date: date
- course_id: int

Candidate Key: {announcement_id}
Primary Key: {announcement_id}
Foreign Key: course_id references course(course_id)

Functional Dependencies:
- announcement_id -> text, date, course_id

Normal Form: BCNF. Because announcement_id is a superkey.

Table Definition:

```
CREATE TABLE announcement(
        announcement _id INT PRIMARY KEY AUTO_INCREMENT,
        text VARCHAR(500) NOT NULL,
        date DATE,
        course_id INT,
        FOREIGN KEY (course_id) REFERENCES course(course_id)
        ON DELETE CASCADE
        ON UPDATE RESTRICT
) ENGINE = INNODB;
```

discount

When the end date is expired, discount will automatically delete the discount offer from the discount entity with an event.
Relational Schema: discount(discount_id, is_allowed, start_date, end_date, percentage, discounted_course_id, allower_course_creator_id)

Attribute Domains:

- discount_id: int
- is_allowed: boolean
- start_date: Date
- end_date: Date
- percentage: int
- discounted_course_id: int
- allower_course_creator_id: int

Candidate Keys: {discount_id}, {discounted_course_id}
Primary Key: {discount_id}
Foreign Key:

- discounted_course_id references course(course_id)
- allower_course_creator _id references course_creator(course_creator_id)

Functional Dependencies:

- discount_id -> is_allowed, start_date, end_date, percentage

Normal Form: BCNF. Because discount_id is a superkey.

Table Definition:

```
CREATE TABLE discount(
        discount _id INT PRIMARY KEY,
        is_allowed BOOLEAN,
```

```
        start_date DATE,
        end_date DATE,
        percentage INT,
        FOREIGN KEY (discounted_course_id) REFERENCES course(course_id)
        ON DELETE CASCADE
        ON UPDATE RESTRICT,
        FOREIGN KEY (allower_course_creator_id) REFERENCES
course_creator(course_creator_id)
        ON DELETE CASCADE
        ON UPDATE RESTRICT,
        CONSTRAINT check_percentage_validity CHECK (percentage > 0 AND percentage <=
100)
) ENGINE = INNODB;
```

refund

Relational Schema: refund(refund_id, reason, is_assessed, is_approved)

Attribute Domains:

- refund_id: int
- reason: varchar(320)
- is_assessed: boolean
- is_approved: boolean

Candidate Keys: {refund_id}
Primary Key: {refund_id}
Foreign Key: None

Functional Dependencies:

- refund_id -> reason, is_assessed, is_approved

Normal Form: BCNF. Because refund_id is a superkey.

Table Definition:

```
CREATE TABLE refund(
        refund_id INT PRIMARY KEY,
        reason VARCHAR(320) NOT NULL,
        is_assessed BOOLEAN DEFAULT FALSE,
        is_approved BOOLEAN DEFAULT FALSE,
) ENGINE = INNODB;
```

refund_requests

Relational Schema: refund_requests(refund_id, student_id, course_id)

Attribute Domains:
- refund_id: int
- student_id: int
- course_id: int

Candidate Keys: {refund_id}, {student_id, course_id}
Primary Key: {refund_id}
Foreign Keys:
- refund_id references refund(refund_id)
- student_id references student(student_id)
- course_id references course(course_id)

Functional Dependencies:
- refund_id -> student_id, course_id
- student_id, course_id -> refund_id

Normal Form: BCNF. Because both refund_id and {student_id, course_id} are superkeys.

Table Definition:

```
CREATE TABLE request_refunds(
        refund_id INT PRIMARY KEY,
        student_id INT,
        course_id INT,
        FOREIGN KEY (refund_id) REFERENCES refund(refund_id)
        ON DELETE CASCADE
        ON UPDATE RESTRICT,
        FOREIGN KEY (student_id) REFERENCES student(student_id)
        ON DELETE CASCADE
        ON UPDATE RESTRICT,
        FOREIGN KEY (course_id) REFERENCES course(course_id)
        ON DELETE CASCADE
        ON UPDATE RESTRICT

) ENGINE = INNODB;
```

complaint

Relational Schema: complaint(complaint_id, complaint_note, complaint_date)

Attribute Domains:
- complaint_id: int
- complaint_note: varchar(360)
- complaint_date: date

Candidate Key: {complaint_id}
Primary Key: {complaint_id}
Foreign Key: None

Functional Dependencies:
- complaint_id -> complaint_note, complaint_date

Normal Form: BCNF. Because complaint_id is a superkey.

Table Definition:

```
CREATE TABLE complaint (
        complaint_id INT PRIMARY KEY AUTO_INCREMENT,
        complaint_note VARCHAR(360) NOT NULL,
        complaint_date DATE NOT NULL,
) ENGINE = INNODB;
```

feedback

Relational Schema: feedback(feedback_id, feedback_note, rating)

Attribute Domains:
- feedback_id: int
- feedback_note: varchar(360)
- rating: numeric(2, 1)

Candidate Key: {feedback_id}
Primary Key: {feedback_id}
Foreign Key: None

Functional Dependencies:
- feedback_id -> feedback_note, rating

Normal Form: BCNF. Because feedback_id is a superkey.

Table Definition:

```
CREATE TABLE feedback(
        feedback_id INT PRIMARY KEY AUTO_INCREMENT,
        feedback_note VARCHAR(360) NOT NULL,
        rating NUMERIC(2,1) DEFAULT 0,
        CONSTRAINT check_rating CHECK (rating > 0)
) ENGINE = INNODB;
```

certificate

Relation Schema: certificate(certificate_id, date, text)

Attribute Domains:

- certificate_id: int
- date: date
- text: varchar(120)

Candidate Key: {certificate_id}
Primary Key: {certificate_id}
Foreign Key: None

Functional Dependencies:

- certificate_id -> date, text

Normal Form: BCNF. Because certificate_id is a superkey.

Table Definition:

```
CREATE TABLE certificate (
        certificate_id INT PRIMARY KEY AUTO_INCREMENT,
        date DATE NOT NULL,
        text VARCHAR(120) NOT NULL
) ENGINE = INNODB;
```

student_complaints

Relational Schema: student_complaints(complaint_id, student_id, course_id,)

Attribute Domains:

- student_id: int
- course_id: int
- complaint_id: int

Candidate Key: {complaint_id}
Primary Key: {complaint_id}
Foreign Keys:

- student_id references student(student_id)
- course_id references course(course_id)
- complaint_id references complaint(complaint_id)

Functional Dependencies:

- complaint_id -> student_id, course_id

Normal Form: BCNF. Because complaint_id is a superkey.

Table Definition:

```
CREATE TABLE student_complaints (
        complaint_id INT PRIMARY KEY,
        student_id INT,
        course_id INT,
        FOREIGN KEY (student_id) REFERENCES student
        ON DELETE CASCADE
        ON UPDATE RESTRICT,
        FOREIGN KEY (course_id) REFERENCES course
        ON DELETE CASCADE
        ON UPDATE RESTRICT,
        FOREIGN KEY (complaint_id) REFERENCES complaint
        ON DELETE CASCADE
        ON UPDATE RESTRICT,
) ENGINE INNODB;
```

student_feedbacks

Relational Schema: student_feedbacks(feedback_id, student_id, course_id)

Attribute Domains:

- student_id: int
- course_id: int
- feedback_id: int

Candidate Keys: { feedback_id }, { student_id, course_id }
Primary Key: { student_id, course_id }

Foreign Keys:

- student_id references student(student_id)
- course_id references course(course_id)

- feedback_id references feedback(feedback_id)

Functional Dependencies:
- student_id, course_id -> feedback_id
- feedback_id -> student_id, course_id

Normal Form: BCNF. Because both { student_id, course_id } and feedback_id are superkeys.

Table Definition:

```
CREATE TABLE student_feedbacks (
        feedback_id INT UNIQUE,
        student_id INT,
        course_id INT,
        PRIMARY KEY (student_id, course_id),
        FOREIGN KEY (student_id) references student
                                ON UPDATE RESTRICT,
        FOREIGN KEY (course_id) references course
                                ON UPDATE RESTRICT,
        FOREIGN KEY (feedback_id) references feedback
                                ON UPDATE RESTRICT
) ENGINE = INNODB;
```

earns

Relational Schema: earns(<u>student_id</u>, <u>course_id</u>, certificate_id)

Attribute Domains:
- student_id: int
- course_id: int
- certificate_id: int

Candidate Keys: {student_id, course_id}, certificate_id
Primary Key: { student_id, course_id }
Foreign Keys:
- student_id references student(student_id)
- course_id references course(course_id)
- certificate_id references certificate(certificate_id)

Functional Dependencies:
- student_id, course_id -> certificate_id
- certificate_id -> student_id, course_id

Normal Form: BCNF. Because both { student_id, course_id } and certificate_id are superkeys.

Table Definition:

```
CREATE TABLE earns(
        student_id INT,
        course_id INT,
        certificate_id INT UNIQUE,
        PRIMARY KEY (student_id, course_id),
        FOREIGN KEY (student_id) REFERENCES student
        ON UPDATE RESTRICT,
        FOREIGN KEY (course_id) REFERENCES course
        ON UPDATE RESTRICT,
        FOREIGN KEY (certificate_id) REFERENCES certificate
        ON UPDATE RESTRICT
) ENGINE = INNODB;
```

note

Relational Schema: note(note_id, student_id, lecture_id, course_id, note_text)

Attribute Domains:

- note_id: int
- student_id: int
- lecture_id: int
- course_id int
- note_text: varchar(360)

Candidate Key: {note_id}
Primary Key: {note_id}
Foreign Keys:

- student_id from student(student_id)
- (lecture_id, course_id) from lecture(lecture_id, course_id)

Functional Dependencies:

- note_id -> student_id, lecture_id, course_id, note_text

Normal Form: BCNF. Because note_id is a superkey.

Table Definition:

```
CREATE TABLE note(
        note_id INT PRIMARY KEY AUTO_INCREMENT,
        student_id INT,
        lecture_id INT,
```

```
    course_id INT,
    note_text VARCHAR(360) NOT NULL,
    FOREIGN KEY (student_id) REFERENCES student
    FOREIGN KEY (lecture_id, course_id) REFERENCES lecture
) ENGINE = INNODB;
```

asks

Relational Schema: asks(question_id, student_id, course_id)

Attribute Domains:

- question_id: int
- student_id: int
- course_id: int

Candidate Key: {question_id}
Primary Keys: {question_id}
Foreign Keys:

- student_id references student(student_id)
- course_id references course(course_id)
- question_id references question(question_id)

Functional Dependencies:

- question_id -> student_id, course_id

Normal Form: BCNF. Because question_id is a superkey.

Table Definition:

```
CREATE TABLE asks(
    question_id INT,
    student_id INT,
    course_id INT,
    PRIMARY KEY ( question_id ),
    FOREIGN KEY (student_id) REFERENCES student
    ON DELETE CASCADE
    ON UPDATE RESTRICT,
    FOREIGN KEY (course_id) REFERENCES course
    ON DELETE CASCADE
    ON UPDATE RESTRICT,
    FOREIGN KEY (question_id) REFERENCES question
    ON DELETE CASCADE
    ON UPDATE RESTRICT
) ENGINE = INNODB;
```

answers

Relational Schema: answers(question_id, course_creator_id, answer_text)

Attribute Domains:

- question_id: int
- course_creator_id: int
- answer_text: varchar(360)

Candidate Key: {question_id}
Primary Key: {question_id}
Foreign Keys:

- question_id references question(question_id)
- course_creator_id references course_creator(course_creator_id)

Functional Dependencies:

- question_id -> course_creator_id, answer_Text

Normal Form: BCNF. Because question_id is a superkey.

Table Definition:

```
CREATE TABLE answers(
        question_id INT PRIMARY KEY,
        course_creator_id INT,
        answer_text VARCHAR(360) NOT NULL,
        FOREIGN KEY (question_id) references question
        ON DELETE CASCADE
        ON UPDATE RESTRICT,
        FOREIGN KEY (course_creator_id) references course_creator
        ON DELETE CASCADE
        ON UPDATE RESTRICT
) ENGINE = INNODB;
```

User Interface Design & Corresponding SQL Statements

Sign Up

Sign Up as a Student

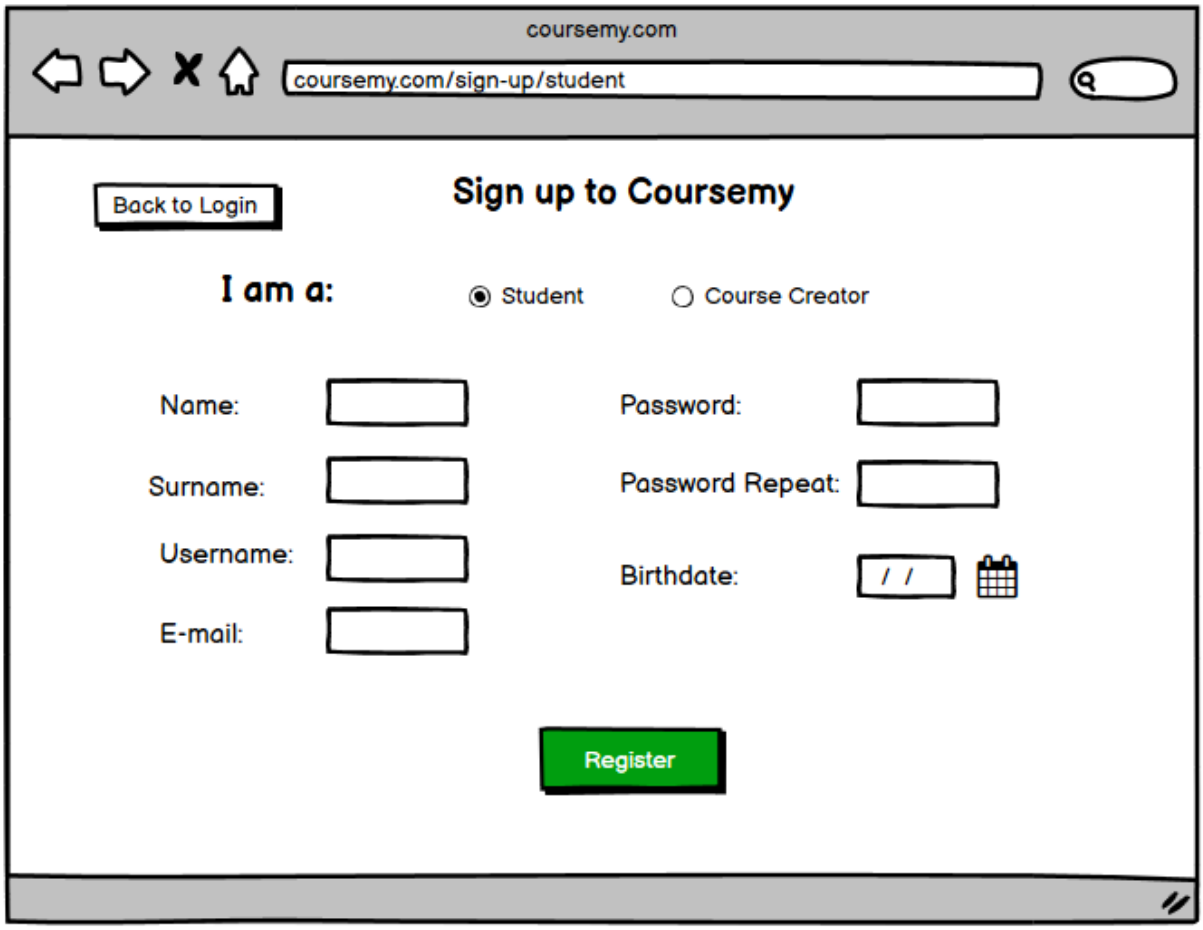

Fig.2: Sign up page for student

Same email address cannot be used for more than one account. All usernames should be unique, too. Thus, if username and/or email is not unique upon pressing the Register button, the registration is unsuccessful as the SQL insert statement gives an error. In this case, an error message is displayed to the student. If username and email are unique (does not exist in the database), then registration is successful. Thus, the new person and student is added to the database.

SQL statement for adding a new person:

INSERT INTO person (username, email, name, surname, password, date_of_birth)
VALUES (@username, @email, @name, @surname, @password, @birthdate)

After executing the query above, if a pre-existing username or email is entered, the error that is returned is examined and which attribute(s) caused that error is read and a proper error message is shown to the user such as 'Entered email is already in use' for email repetition.

SQL statement for adding a new student:

INSERT INTO student **VALUES** (@person_id, 0)

Sign Up as a Course Creator

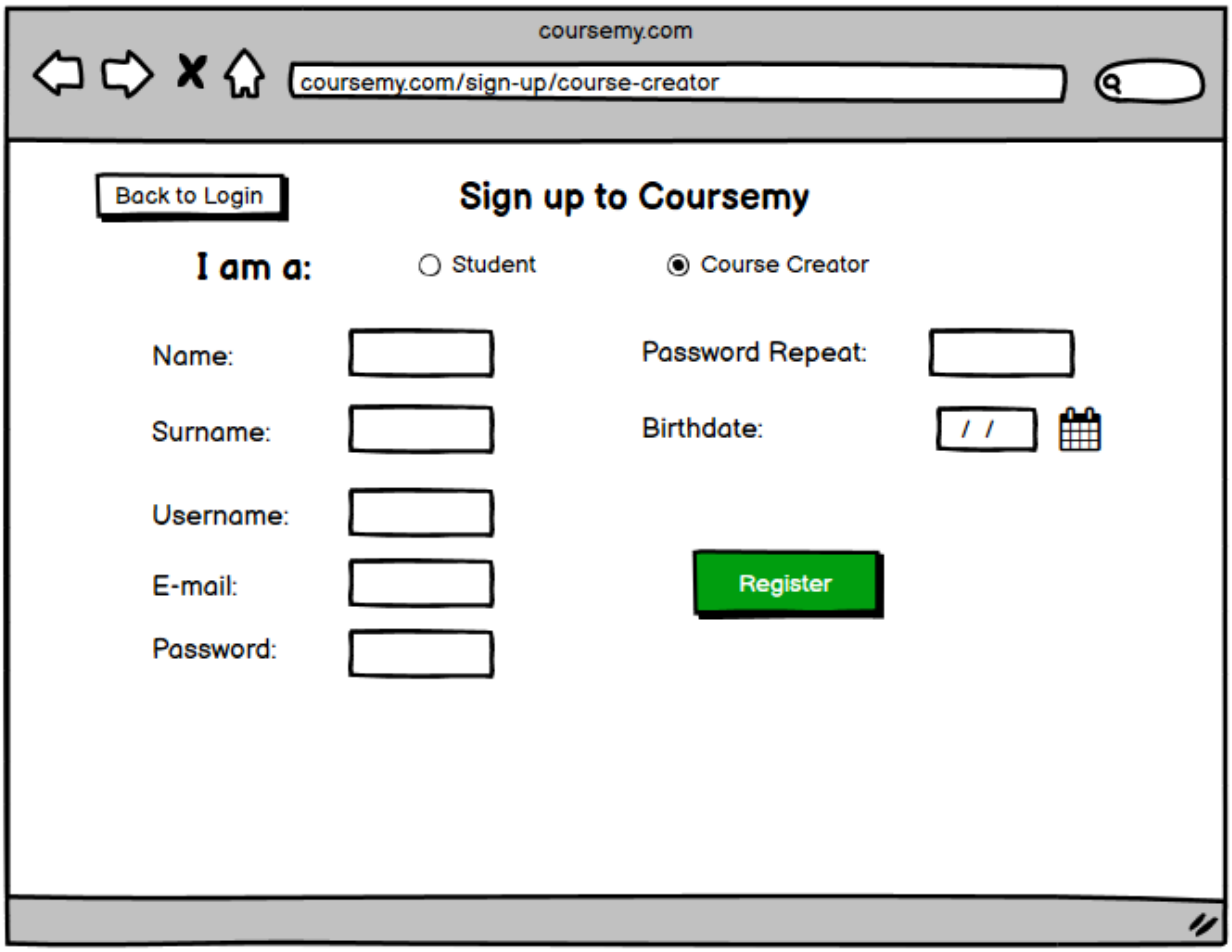

Fig.3: Sign up page for course creator

The same insertion operation while signing up as a student is performed. If email and username are unique, the registration is successful and the new person and the course creator are added to the database.

SQL statement for inserting a person:

INSERT INTO person (username, email, name, surname, password, date_of_birth)
VALUES (@username, @email, @name, @surname, @password, @birthdate)

SQL statement for adding a new course creator:

INSERT INTO course_creator **VALUES** (@person_id, 0, 0)

Login

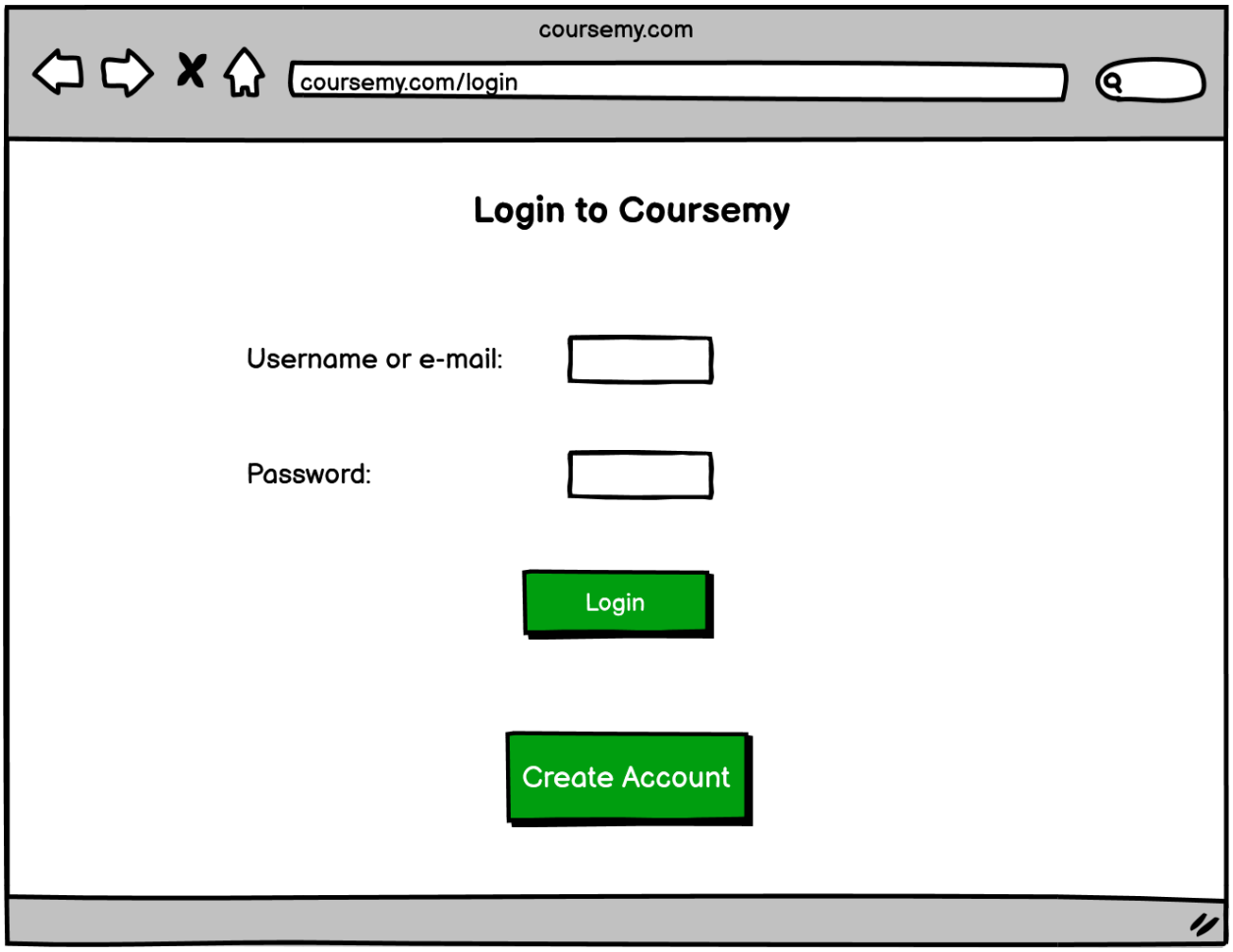

Fig. 4: Login page, common for all types of users

The first executed query after clicking the ‘Login’ button is below for authentication.

SQL statement for searching for credentials with email or username:

```
SELECT    person_id
FROM      person
WHERE     (email = @entered_username_or_email OR
           username = @entered_username_or_email) AND
           password = @entered_password
```

If the query above returns a person_id, this person_id is stored to be used in the application (@person_id). Otherwise, 'username or email does not exist' error is shown to the user.

In order to understand the type of the user, the following queries are executed. According to the type of the user, different UI screens are shown.

```
SELECT    person_id, name, surname
FROM      person P, student S
WHERE     P.person_id = S.student_id AND S.student_id = @person_id
```

If this query returns a nonempty result, then we conclude that the user entered to the website is a student. Else, the following query is executed.

```
SELECT    P.person_id, P.name, P.surname
FROM      person P, course_creator C
WHERE     P.person_id = C.course_creator_id AND
           C.course_creator_id = @person_id
```

If this query returns a nonempty result, then we conclude that the user entered to the website is a course_creator. Else, the following query is executed for admin:

```
SELECT    P.person_id, P.name, P.surname
FROM      person P, admin A
WHERE     P.person_id = A.admin_id AND A.admin_id = @person_id
```

If this query returns a nonempty result, then we conclude that the user entered to the website is an admin.

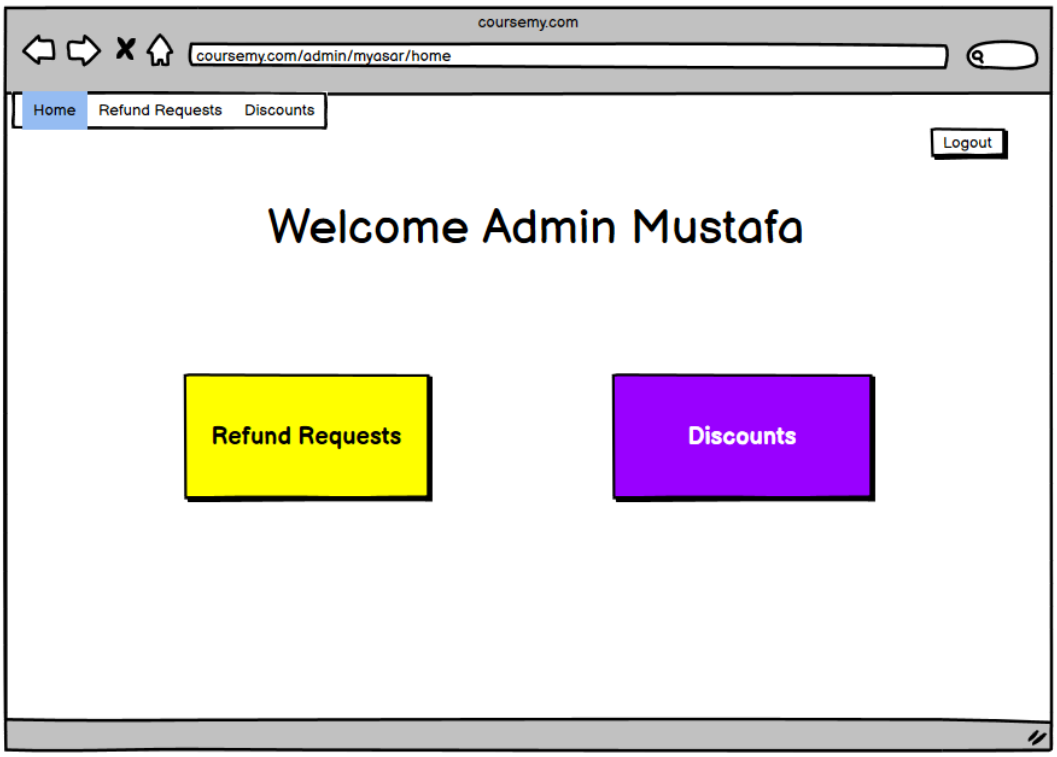

Fig.5: Home page for admin after login

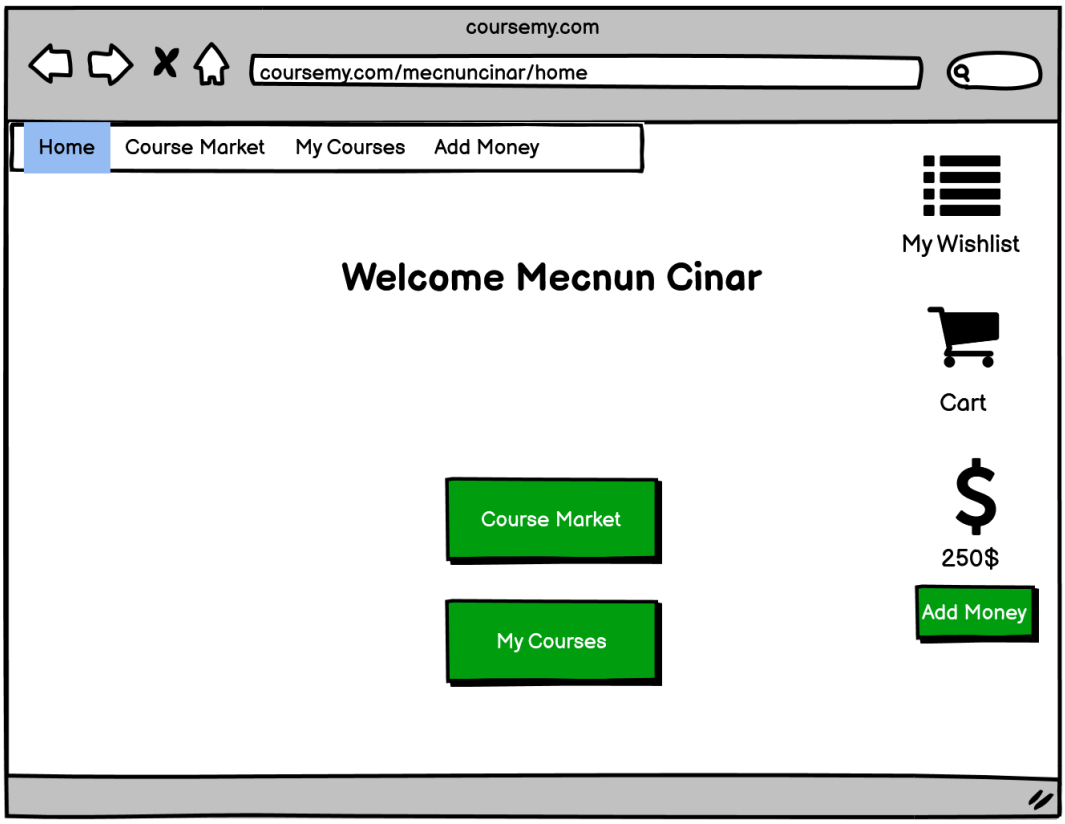

Fig.6: Home page for student after login

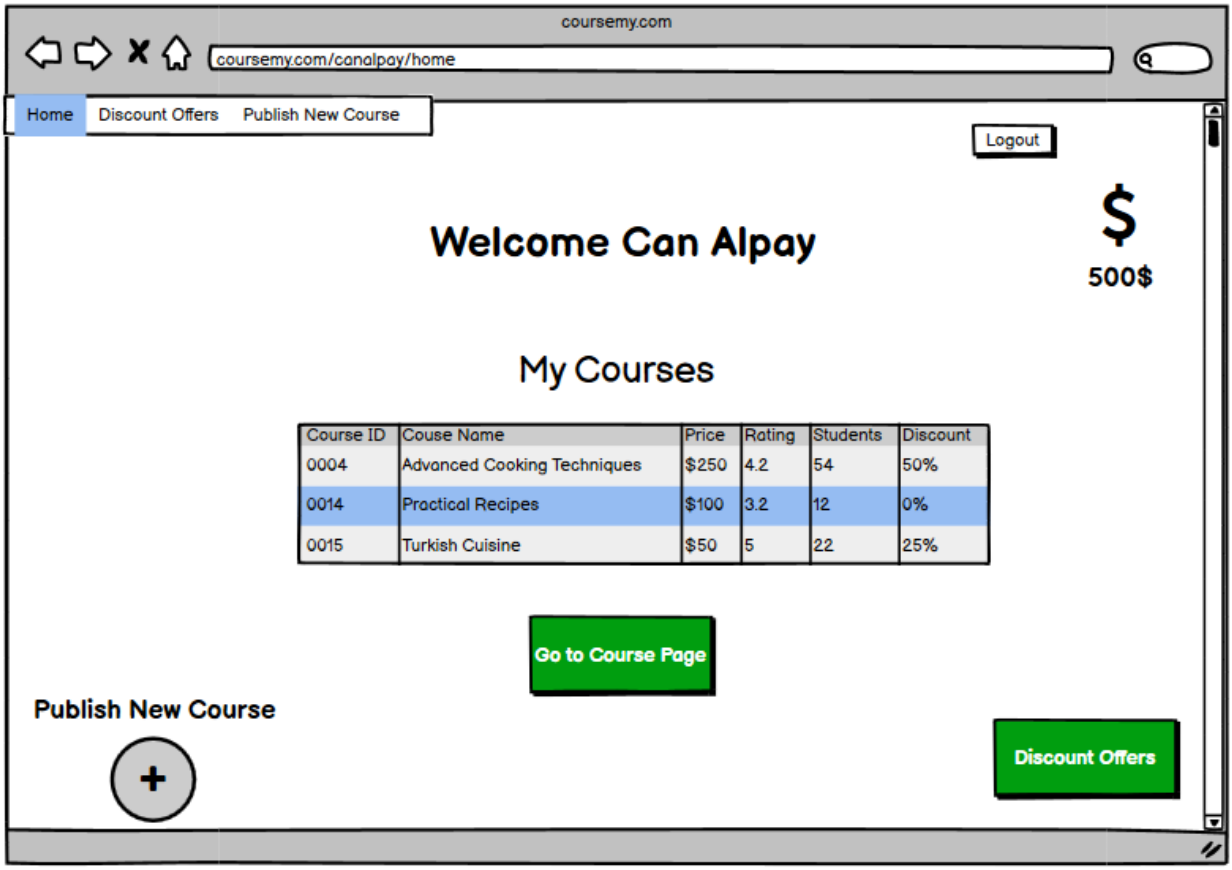

Fig.7: Home page for course creator after login

Additional Functionalities (Assignment Feature)

A. View Assignments of a Course for Student

After clicking on My Courses Page and navigating to the desired course page that is already purchased by the student, the student can see assignments for the course by clicking on the Assignments button on the course page. Then, assignments of a particular course show up on this page along with assignment information like assignment id, threshold and how many attempts the student has taken.

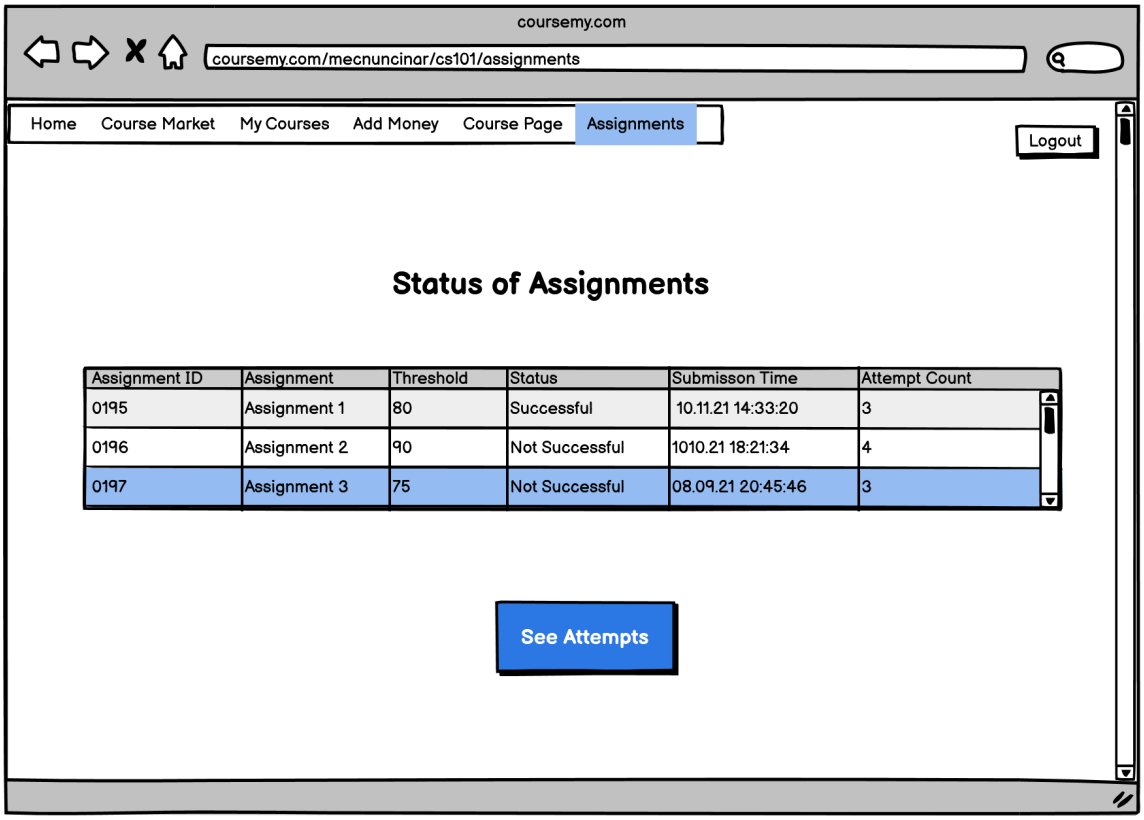

Fig.8: Assignments overview page for students

SQL statement for displaying all assignments of a course with attempts:

```
SELECT    A.assignment_id, A.assignment_threshold, S.attempts
FROM      (SELECT assignment_id, student_id, COUNT(S.submission_time) AS
          Attempts
          FROM submitted_assignments
          GROUP BY (assignment_id, student_id)) S, assignment A
WHERE     A.course_id = @course_id AND A.assignment_id = S.assignment_id AND
          S.student_id = @student_id
```

While creating the table, for every row of the table, the assignment_id is given to the statement below, if this statement returns a nonempty set, we conclude that the status of that assignment is successful, not successful otherwise.

```
SELECT    *
FROM      submitted_assignments
WHERE     assignment_id=@assignment_id AND
          student_id=@student_id AND
          grade >= threshold
```

A student can choose an assignment from this page, click on it to see his/her attempts/submissions for that assignment. Chosen @assignment_id is given to this new page.

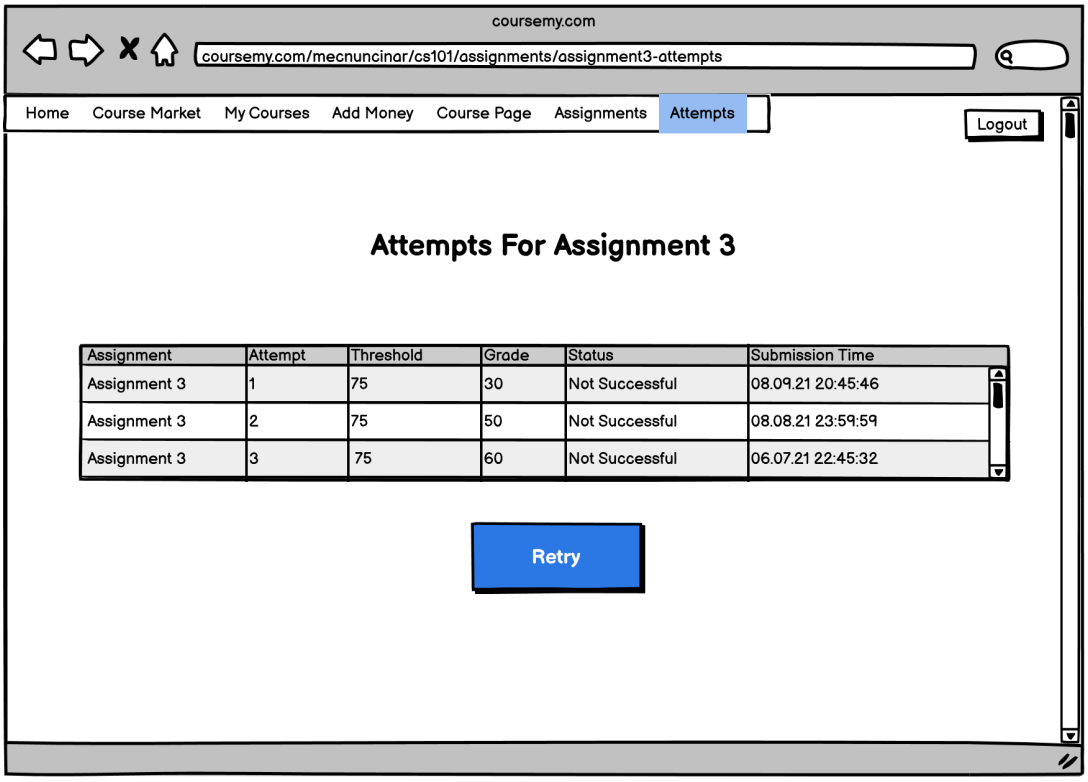

Fig.9: Attempts page for an unsuccessful assignment with Retry button enabled

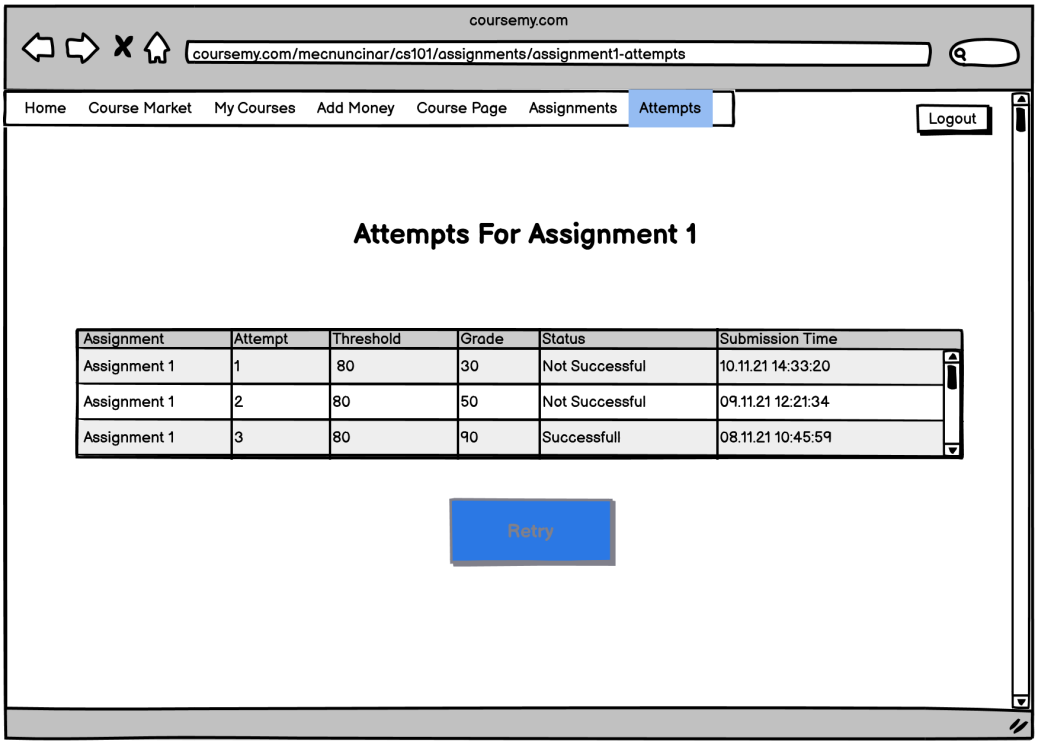

Fig.10: Attempts page for a successful assignment with Retry button disabled

SQL statement for displaying all submissions of a student for an assignment with grades:

```
SELECT      submitted_time, grade
FROM        submitted_assignments
WHERE       student_id = @student_id AND assignment_id = @assignment_id
ORDER BY    submitted_time DESC
```

B. Submit an Assignment Page for Student

On the page where all submissions for an assignment for a student is shown, there is a Retry button to take the assignment again. This button is active if the student has not yet passed the assignment successfully as can be seen from *Fig.9* and *Fig.10*.

SQL statement to check if the student has passed the assignment:

```
SELECT      *
FROM        submitted_assignments
WHERE       assignment_id=@assignment_id AND student_id=@student_id
```

After executing the query above, all of the attempts of the student on the assignment are stored. Every attempt is looped and checked, if a grade in an attempt is greater than or equal to the threshold value, the assignment is considered as passed.

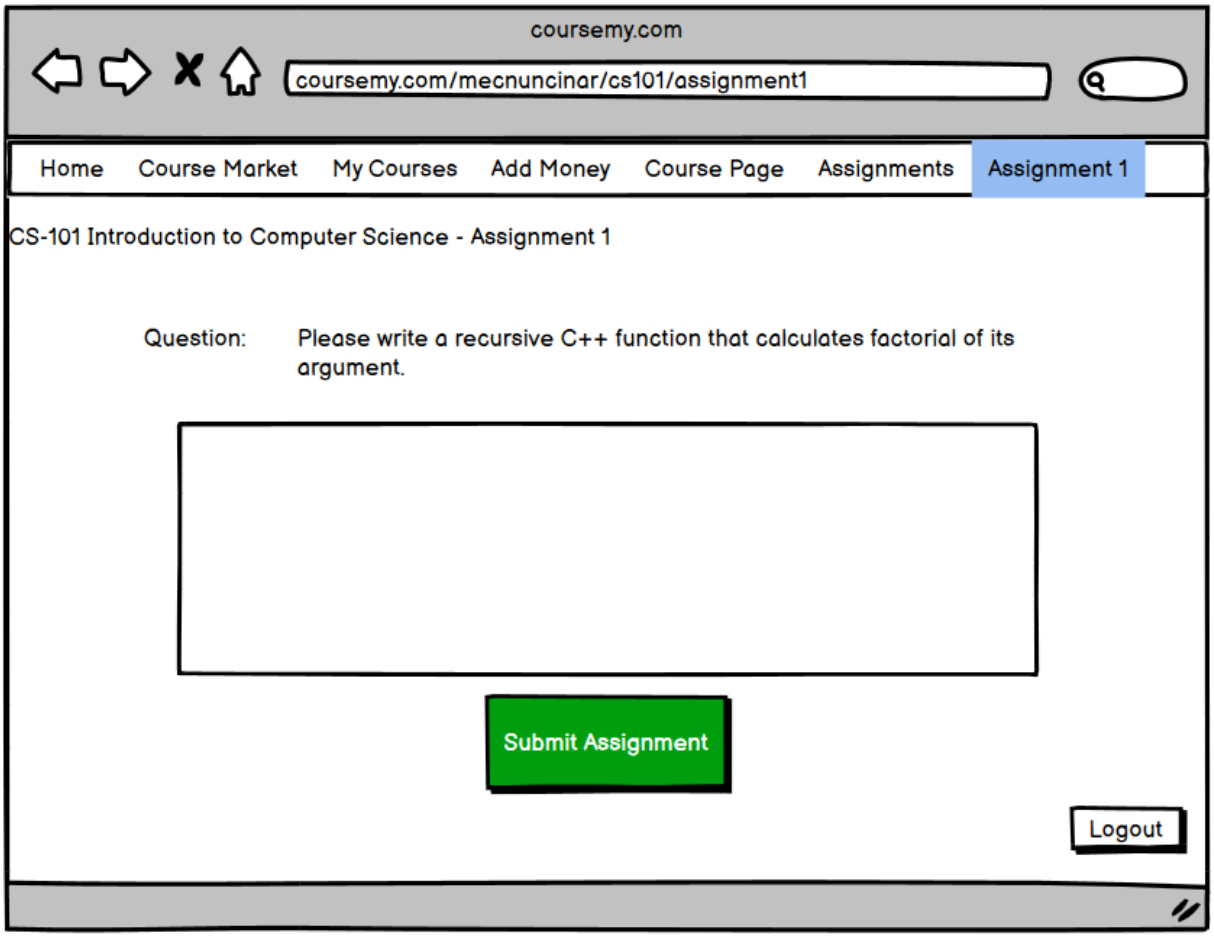

Fig.11: Assignment answering page for a student

If the button is active, the student can take the assignment again. After clicking on the button, an assignment answering page is displayed with a text field for answer input and a text for assignment question above it. After giving an answer (answer should not be null), the student can submit his/her assignment with the following statement:

SQL statement to display assignment question:

```
SELECT      assignment_question
FROM        assignment
WHERE       assignment_id = @assignment_id
```

SQL statement to submit assignment:

```
INSERT INTO submitted_assignments(assignment_id, student_id, submission_time,
        assignment_answer, grade)
VALUES (@assignment_id, @student_id, CURRENT_TIMESTAMP, @answer, NULL)
```

Then, the student is taken back to his/her submissions page again for an assignment.

C. View Submitted Assignments of a Course for Course Creator

The course creator accesses the course page of his/her offered course by clicking Go To Course Page button on the Home screen, and clicks on View Assignments button to open all assignments he/she has put up for the course and sees it in a list.

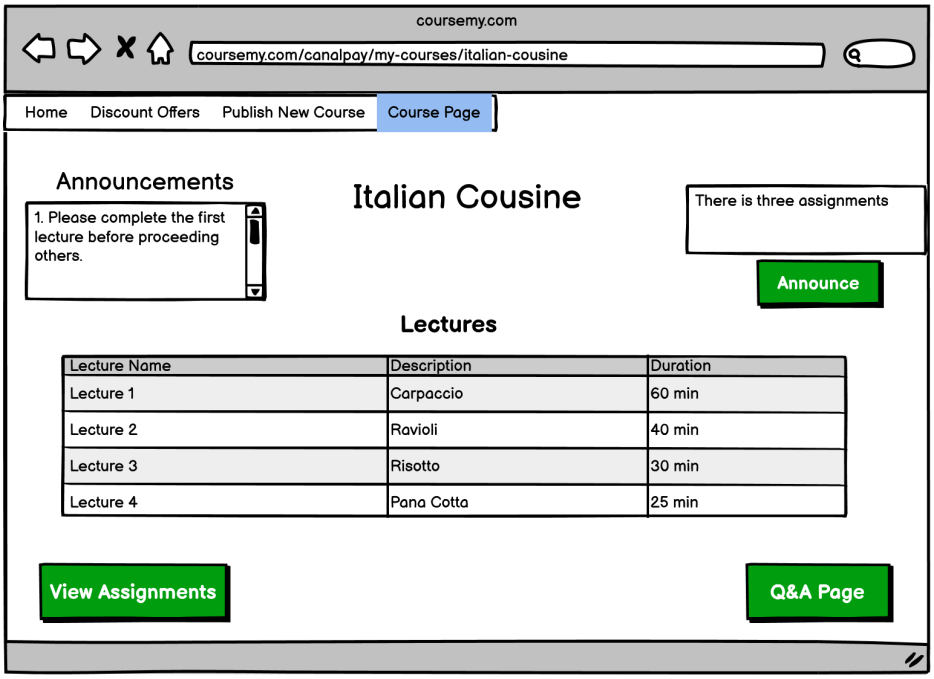

Fig.12: Course page from the perspective of a course creator

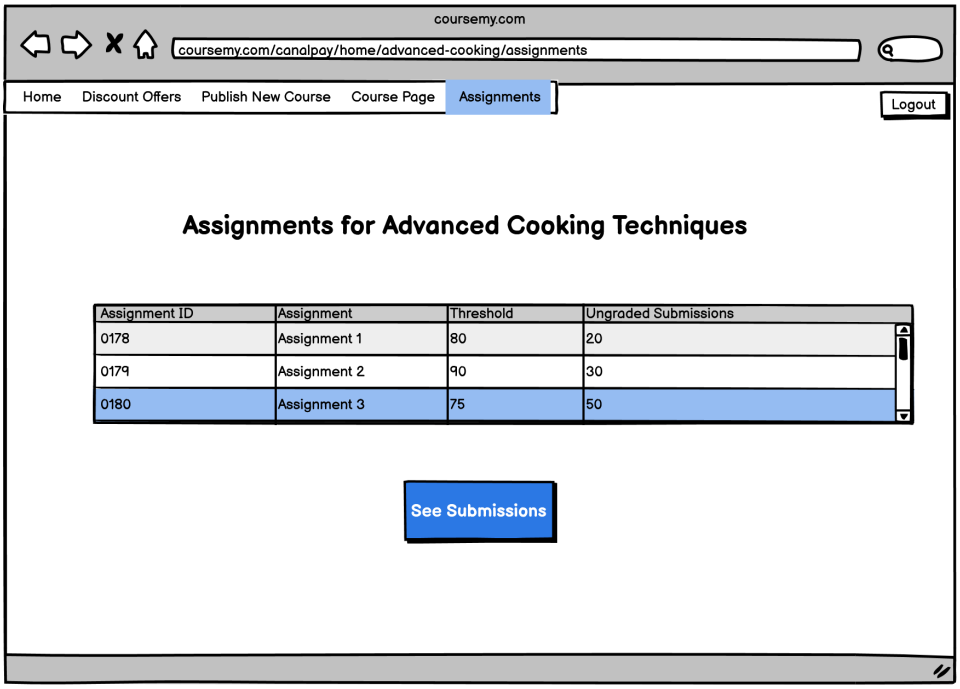

Fig.13: Assignments page from the perspective of a course creator

SQL statement for seeing all assignments for the course along with the count of ungraded submissions of students for that assignment:

```
SELECT    A.assignment_id, A.threshold, asg_to_grade.ungraded_sub_count
FROM      (SELECT assignment_id, COUNT(student_id) AS ungraded_sub_count
          FROM submitted_assignments
          WHERE assignment_id = A.assignment_id AND grade IS NULL
          GROUP BY assignment_id) asg_to_grade, assignment A
WHERE     A.course_id = @course_id
```

To grade an assignment in the submitted assignment pages of a course, the course creator clicks on the assignment he/she wants to grade from this list. Doing so, the program stores the chosen assignment's @assignment_id and passes it to the new page. Here, the course creator can see all submissions for an assignment of a particular course that are waiting to be graded. The submissions display the student full name along with submission details. Full name is concatenated in PHP code.

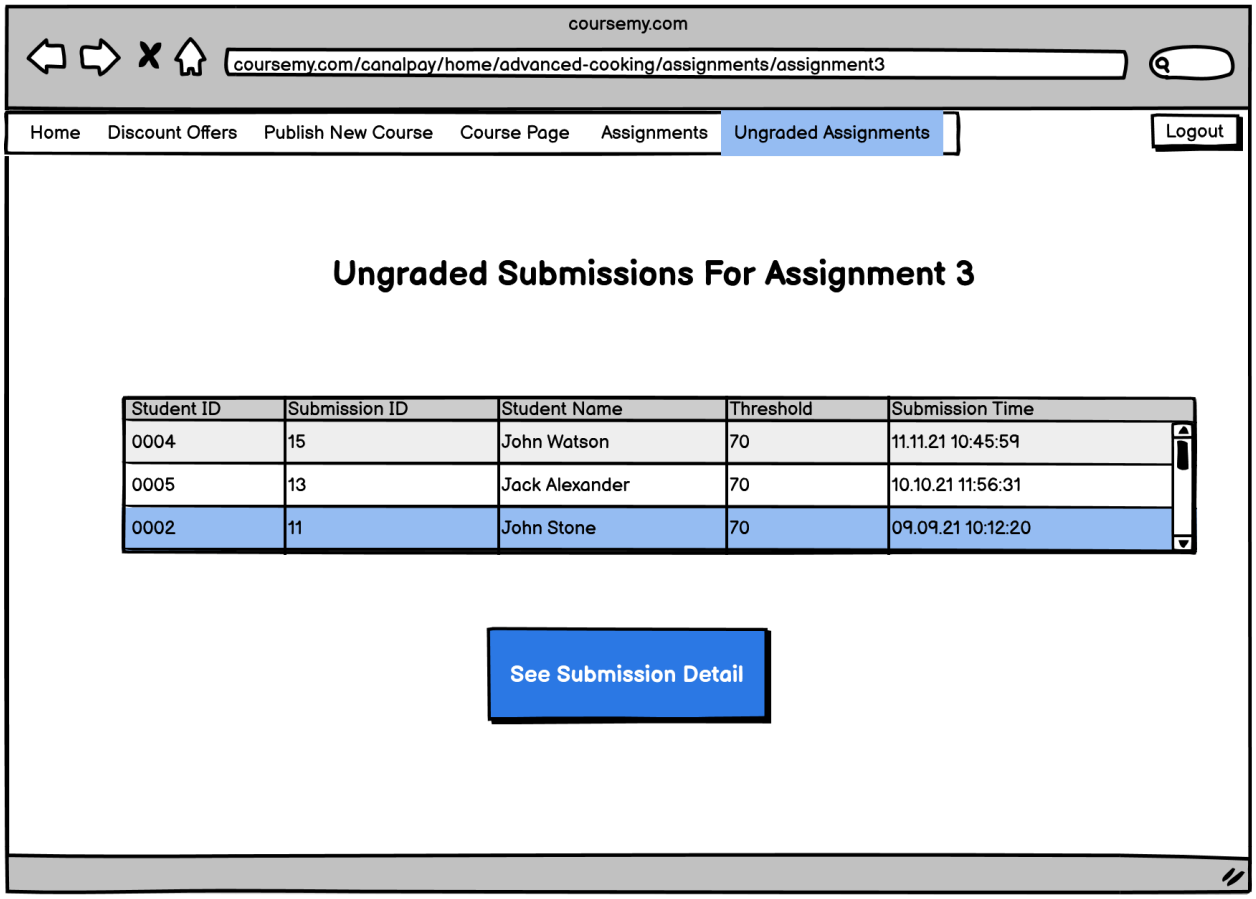

Fig.14: Assignments page from the perspective of a course creator

SQL statement for seeing ungraded submissions for an assignment:

```
SELECT    G.student_id, P.name, P.surname, G.submission_time
FROM      submitted_assignments G, person P
WHERE     grade IS NULL AND assignment_id = @assignment_id AND
          P.id = G.student_id
```

Then, the course creator can choose any assignment submission from this list to see the answer and give a grade.

D. Grading of Assignment by Course Creator

To see the answer given by the student, the course creator clicks on one of the ungraded submissions where the assignment id, student id and submission_time are taken from the previous page upon selection:

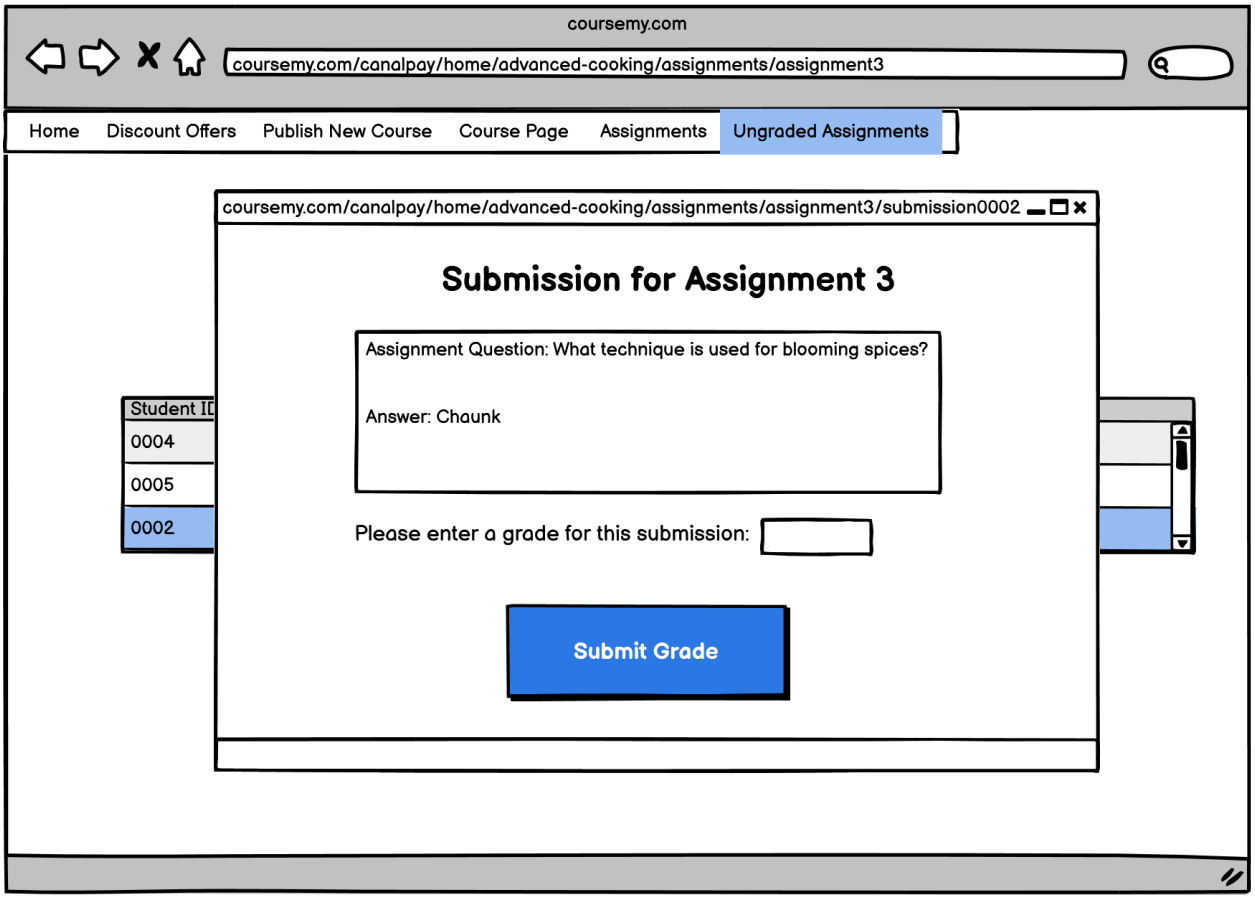

Fig.15: Grading page for an assignment for course creator

SQL statement for seeing the students' ungraded submission along with the question:

```
SELECT    S.assignment_id, S.student_id, S.submission_time,
          S.assignment_answer, A.assignment_question
FROM      submitted_assignments S, assignment A
WHERE     S.assignment_id = @assignment_id AND S.student_id = @student_id
          AND S.submission_time = @submission_time AND
          A.assignment_id = S.assignment_id
```

Then, the course creator can enter a grade on the input field for the assignment:

SQL statement for updating the grade of a submitted assignment:

```
UPDATE    submitted_assignments
SET       grade = @grade
WHERE     assignment_id = @assignment_id AND student_id = @student_id AND
          submission_time = @submission_time AND grade IS NULL
```

Assignment creation process is mentioned in the course publishing process in the report.

Topic Specific Functionalities

I. Buy a Course as Student

A. Course Discover Page

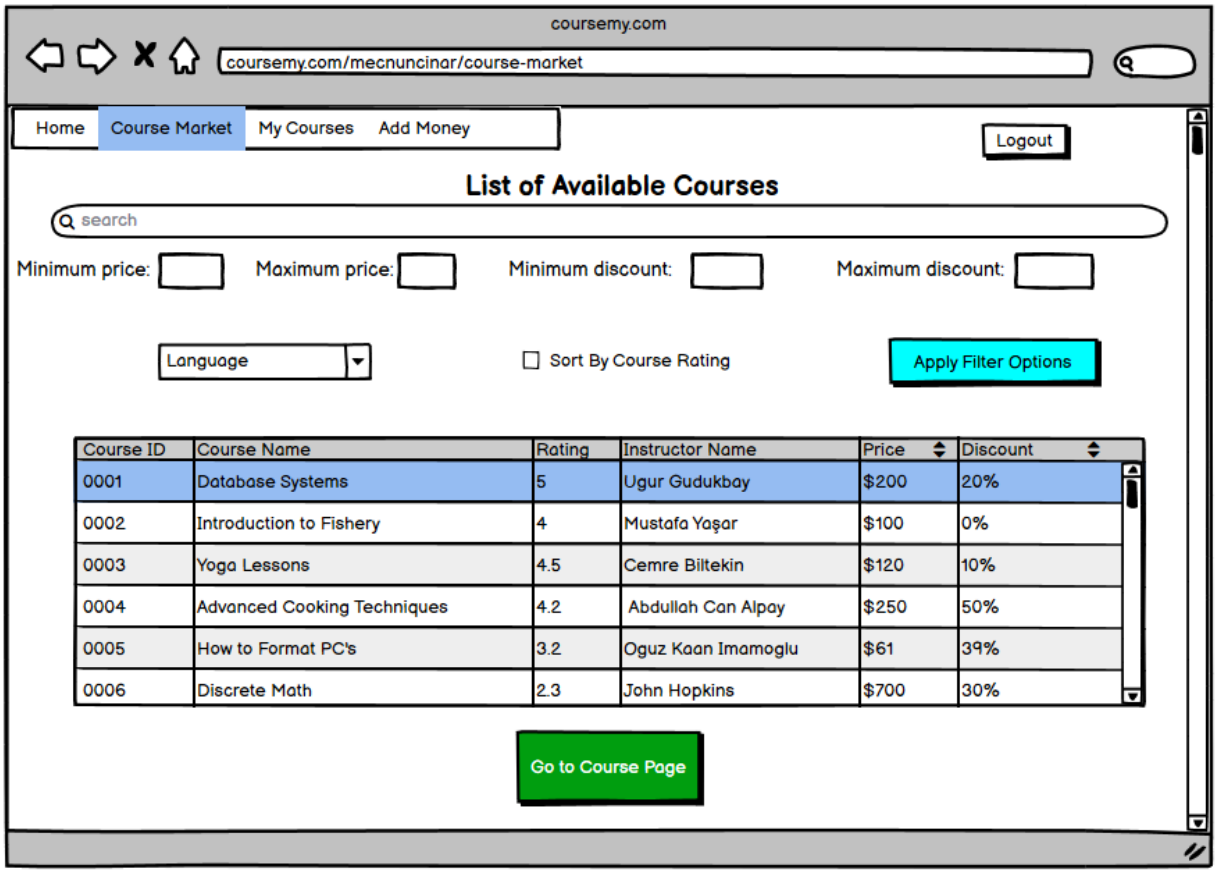

Fig.16: Course market page from a student perspective

SQL statement for all available courses:

SELECT_ALL_COURSES:

```
SELECT    C.course_id, C.course_name, C.language, C.average_rating, C.category,
          P.name, P.surname, D.percentage, D.start_date, D.end_date,
          D.is_allowed, (CASE WHEN CURRENT_DATE =< D.end_date AND
                         CURRENT_DATE >= D.start_date AND D.is_allowed
                         THEN C.course_price * (( 100 - D.percentage ) / 100)
                         ELSE C.course_price END) as price
FROM      course C LEFT OUTER JOIN discount D ON
          C.course_id = D.discounted_course_id LEFT JOIN person P ON
          C.course_creator_id = P.person_id
```

In order to simplify the simplicity and prevent the duplication, the above query will be marked as **SELECT_ALL_COURSES**.

This page displays all courses to the student and the student can filter the displayed courses by selecting filters. The courses can have the following filters:

- From the category dropdown box, the student can select a category to display courses only from that particular category.

  SQL statement for filtering by category:

  ```
  SELECT_ALL_COURSES
  WHERE     category=@selected_category;
  ```

- Using the input boxes for price filtering, the student can select a minimum and maximum price value of his/her choice to filter courses with the price falling for this selected range. The default value of the minimum price value is 0, and the default value of the maximum price value is calculated as follows,

  SQL statement for finding the maximum price value:

  ```
  WITH      all_courses (course_id, course_name, language, rating, category, name,
            surname, percentage, start_date, end_date, is_allowed, price) AS
            (SELECT_ALL_COURSES)

  SELECT    MAX(price)
  FROM      all_courses
  ```

  SQL statement for filtering by minimum and maximum price value:

  ```
  WITH      all_courses (course_id, course_name, language, rating, category, name,
            surname, percentage, start_date, end_date, is_allowed, price) AS
            (SELECT_ALL_COURSES)

  SELECT    *
  FROM      all_courses
  WHERE     price >= @minimum AND
            price <= @maximum;
  ```

- Using the input boxes for discount filtering, the student can select a minimum and maximum discount percentage value of his/her choice to filter courses with the discount percentages falling for this selected range. The default value for the minimum discount percentage is 0, and the default maximum discount percentage is 100.

SQL statement for filtering by minimum and maximum discount percentage:

```
SELECT_ALL_COURSES
WHERE       D.percentage >= @minimum AND
            D.percentage <= @maximum AND
            D.is_allowed = 1
```

- Using the search box, the student can search a keyword in course names and course creator full names, and the result is the filtered courses with this keyword. The entered keyword is searched through both course names and course creators in the system, and the result is a union of this search.

```
WITH        all_courses (course_id, course_name, language, rating, category, name,
            surname, percentage, start_date, end_date, is_allowed, price) AS
            (SELECT_ALL_COURSES)

SELECT      *
FROM        all_courses
WHERE       course_name LIKE '%@keyword%' OR
            name LIKE '%@keyword%' OR
            surname LIKE '%@keyword%';
```

- Using the language dropdown menu, the student can filter courses by the language used in the course.

SQL statement for filtering by language:

```
SELECT_ALL_COURSES
WHERE       C.language = @language
```

- Using the sort by course rating checkbox, the student can sort the courses from the best rating (greater rating) to worst rating.

SQL statement for sorting by rating:

```
SELECT_ALL_COURSES
ORDER BY    C.average_rating DESC;
```

In order to combine the filters, in PHP we create a string that is composed of the SQL statement that selects all courses. If the user selects a filter, we add the “WHERE” clause of that query to the string and execute the combined query. Therefore, we can combine the filters.

B. Course Page

The student can select a course from the course market on *Fig.16* to view its features such as rating, price, feedback etc.

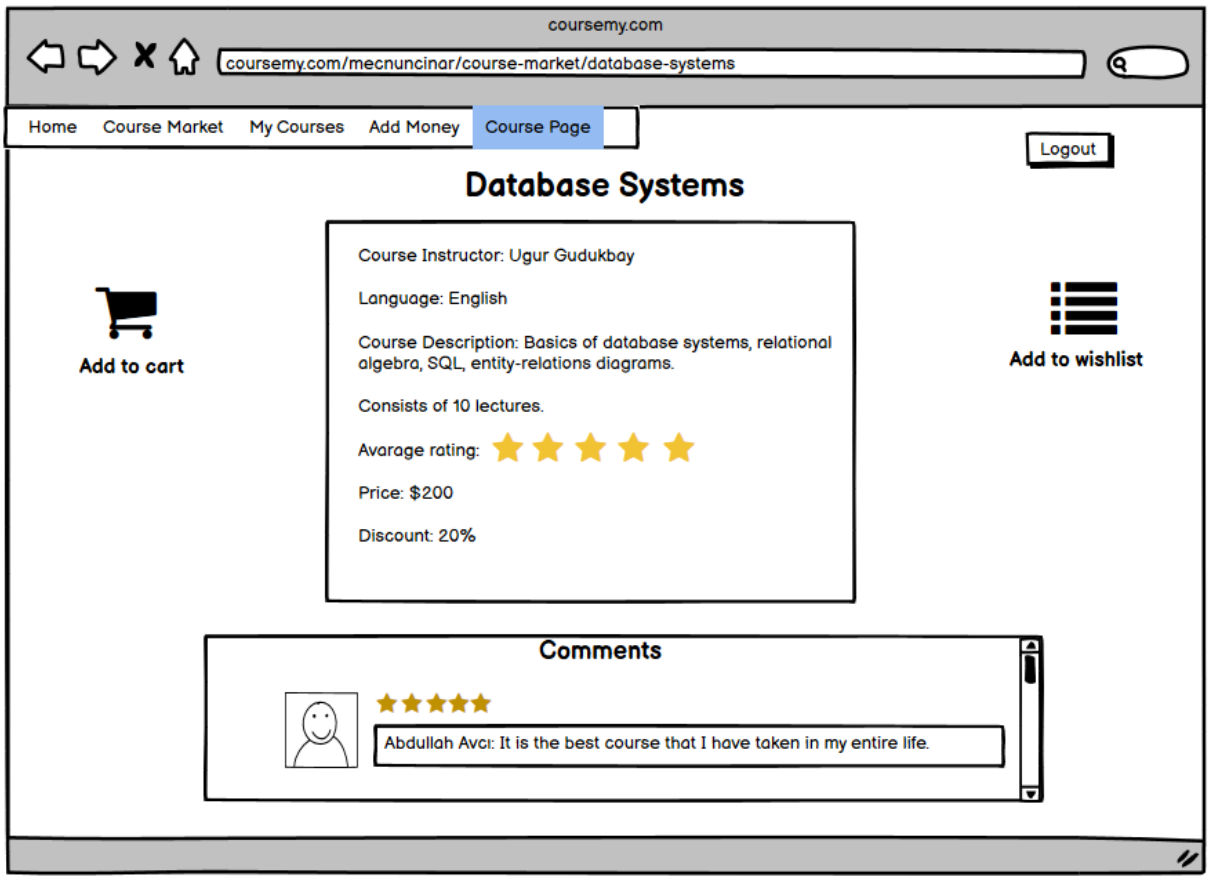

Fig.17: Course page entered from course market

This page displays the selected course's page and displays the course's features in detail.

SQL statement for displaying the selected course:

```
SELECT    C.course_name, C.course_price, C.create_date, C.language
          C.course_description, C.average_rating, P.name, P.surname
FROM      course C, person P
WHERE     C.course_creator_id = P.person_id AND C.course_id = @course_id
```

The percentage information is taken from the query below, if is_allowed column is 0 or NULL, then there is no discount, therefore, the discount label is set to 0. Otherwise, the percentage is shown to the user. In order to find the discount amount of the selected course, the below query is executed.

SQL statement to find the discount information about the selected course

```
SELECT_ALL_COURSES
WHERE     C.course_id=@course_id
```

SQL statement for displaying the feedbacks of the selected course:

```
SELECT    P.name, P.surname, F.feedback_note
FROM      student_feedbacks SF, feedback F, person P
WHERE     SF.student_id=P.person_id AND
          SF.course_id=@course_id AND
          SF.feedback_id=F.feedback_id
```

C. Buying / Wishlisting a Course and Adding a Course to the Cart

A student can add funds to his/her wallet to purchase a course from the home page by clicking on the Add Money button on the Home page. Then a window appears to get input from the student as loaded funds. The student specifies the amount of money to add to his/her account on the input field. By clicking the Add Money Button, the amount is added to the student's wallet.

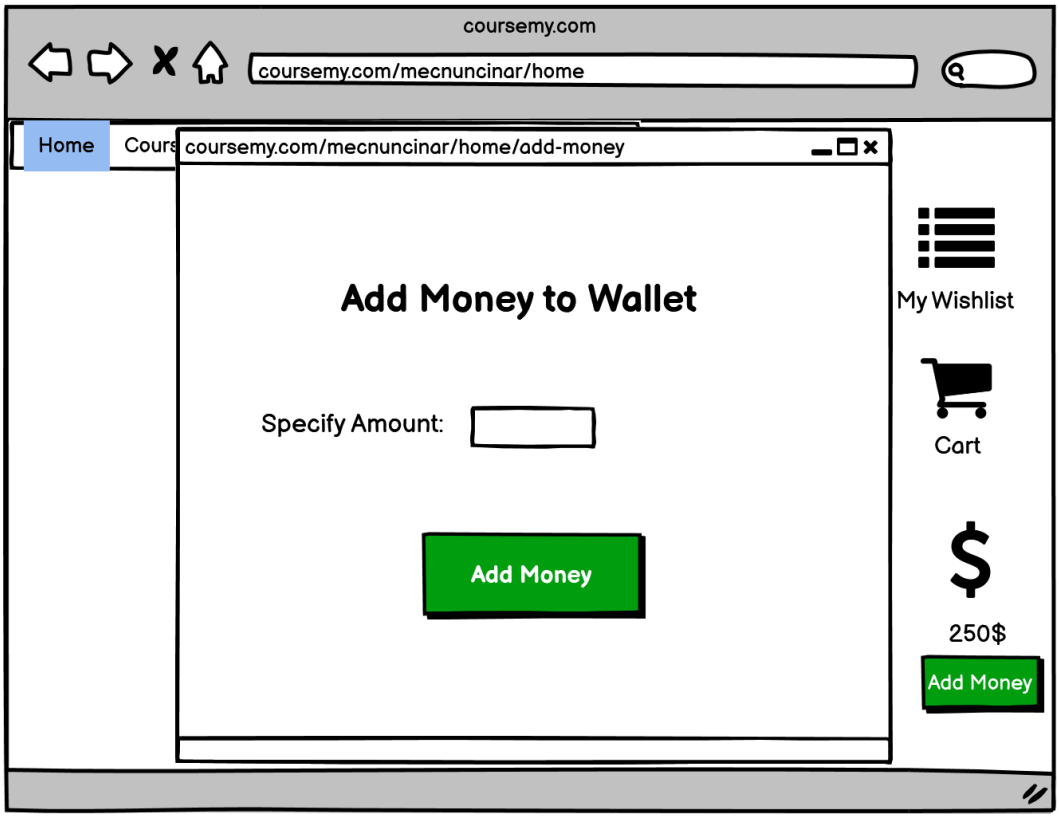

Fig.18: Add money screen for student

SQL statement to add funds to the student wallet:

```
UPDATE      student
SET         wallet = wallet + @added_amount
WHERE       student_id=@student_id
```

SQL statement for buying a course:

First of all, we need to get the price of the course.

```
WITH        all_courses (course_id, course_name, language, rating, category, name,
            surname, percentage, start_date, end_date, is_allowed, price) AS
            (SELECT_ALL_COURSES)

SELECT      price
FROM        all_courses
WHERE       course_id=@course_id
```

The result of the query above is stored in @purchased_price variable and used while inserting it to the enrolls table.

INSERT INTO enrolls(student_id, course_id, purchased_price, purchase_date)
VALUES (@student_id, @course_id, @purchased_price, **CURRENT_DATE**)

After inserting it to the enrolls, the @purchased_price is deducted from the wallet of the student.

SQL statement for reducing the price from the student's wallet:

UPDATE student
SET wallet = wallet - @purchased_price
WHERE student_id=@student_id

After enrolling in the course, the @purchased_price is added to the course_creator's wallet. In order to do this, first, the id of the course creator of the course is fetched.

SQL statement for finding the id of the course creator:

SELECT course_creator_id
FROM course
WHERE course_id=@course_id

After executing the statement above, the result is stored in the @course_creator_id variable and used for the next query.

SQL statement for adding the price to the course_creator's wallet:

UPDATE course_creator
SET wallet = wallet + @purchased_price
WHERE course_creator_id = @course_creator_id

Current date can be achieved by various ways such as using GETDATE() or current_date.

After buying a course, if that course exists in the wishlist and/or cart, that course is removed from the wishlist and/or cart.

SQL statement for removing the bought course from wishlist of the student:

DELETE FROM adds_to_wishlist
WHERE student_id=@student_id **AND** course_id=@course_id

SQL statement for removing the bought course from cart of the student:
DELETE FROM adds_to_cart
WHERE student_id=@student_id **AND** course_id=@course_id

SQL statement for wishlisting a course:

```
INSERT INTO        adds_to_wishlist(student_id, course_id)
VALUES (@student_id, @course_id)
```

SQL statement for adding a course to the cart:

```
INSERT INTO        adds_to_cart(student_id, course_id)
VALUES (@student_id, @course_id)
```

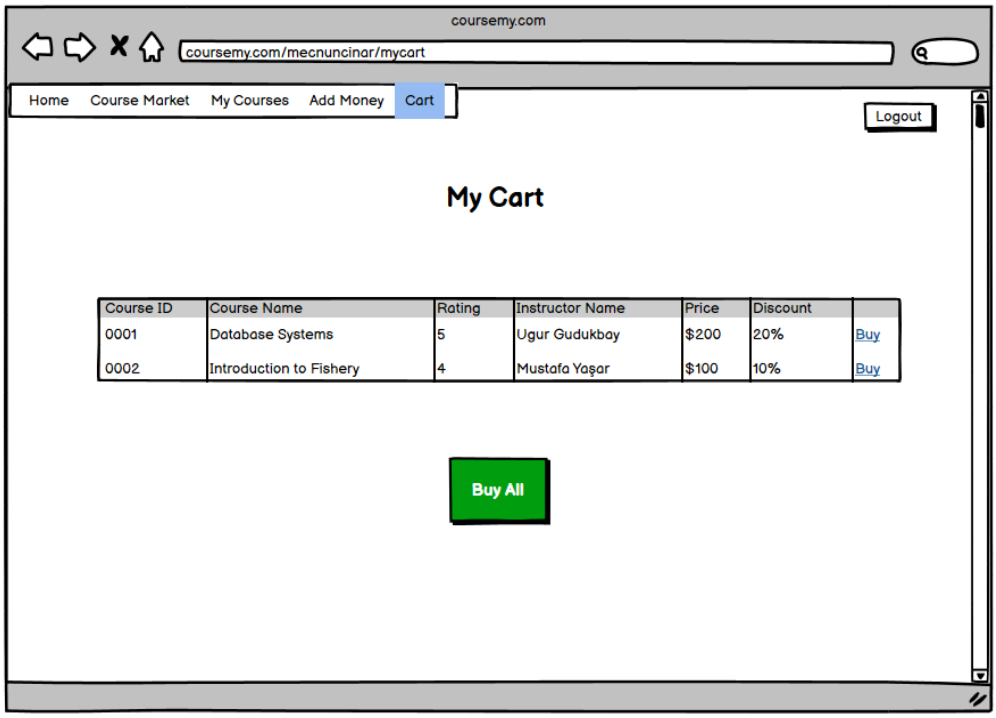

Fig.19: Shopping cart page for student

SQL statement for displaying the cart:

```
SELECT      C.course_name, C.course_price, C.average_rating,
            P.name, P.surname, D.percentage,
            (CASE WHEN CURRENT_DATE < D.end_date AND
                  CURRENT_DATE >= D.start_date AND D.is_allowed
                  THEN C.course_price * (( 100 - D.percentage ) / 100)
                  ELSE c.course_price END) as price
FROM        course C LEFT OUTER JOIN discount D ON
            C.course_id = D.discounted_course_id LEFT JOIN person P ON
            C.course_creator_id = P.person_id LEFT JOIN adds_to_cart AC
            ON P.person_id=AC.student_id
WHERE       C.course_id=AC.course_id    AND
            AC.student_id=@student_id
```

When the student clicks on the My Wishlist, the courses in their wishlist are displayed as the image below.

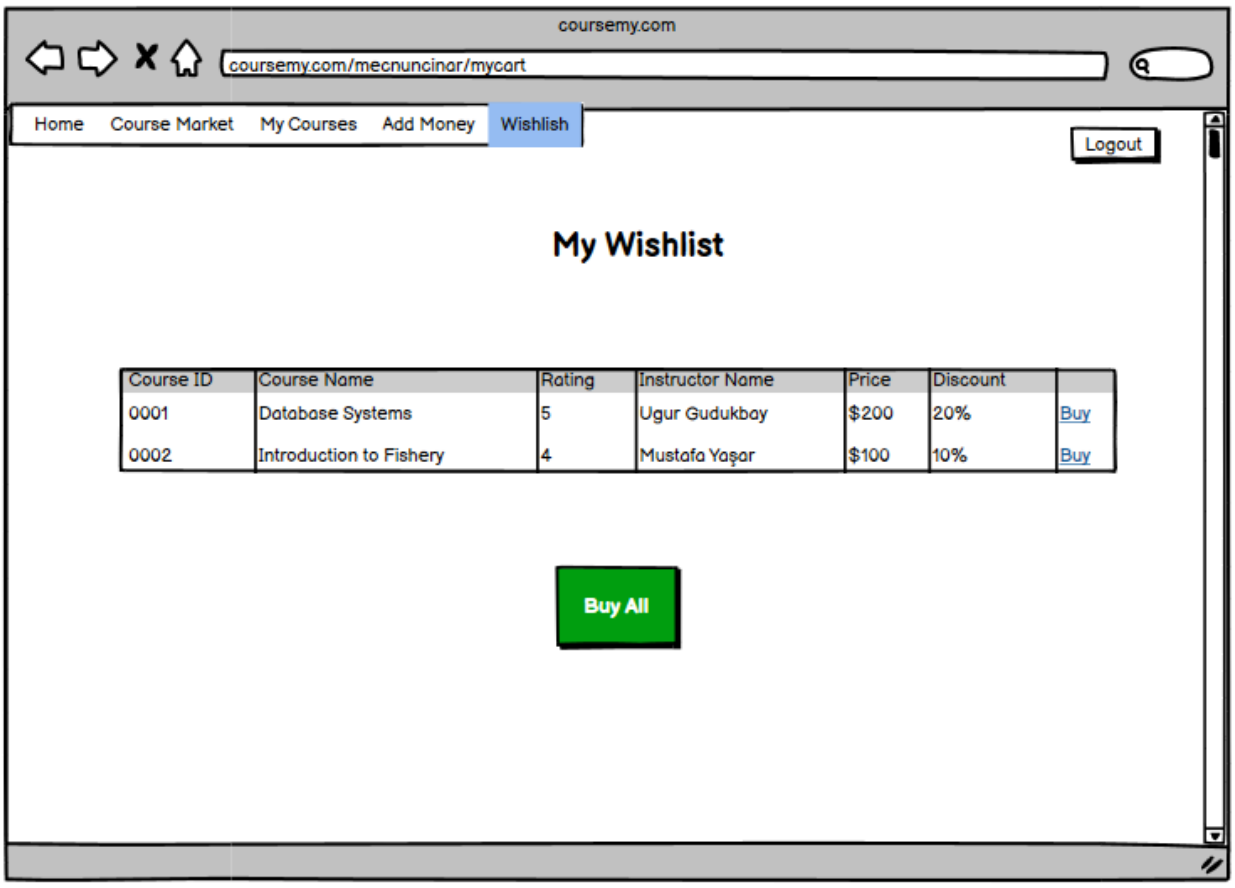

Fig.20: Wishlist screen for student

SQL statement for displaying the wishlist:

```
SELECT    C.course_name, C.course_price, C.average_rating,
          P.name, P.surname, D.percentage,
          (CASE WHEN CURRENT_DATE < D.end_date AND
                     CURRENT_DATE >= D.start_date AND D.is_allowed
                THEN C.course_price * (( 100 - D.percentage ) / 100)
                ELSE c.course_price END) as price
FROM      course C LEFT OUTER JOIN discount D ON
          C.course_id = D.discounted_course_id LEFT JOIN person P ON
          C.course_creator_id = P.person_id LEFT JOIN adds_to_wishlist AW ON
          P.person_id=AW.student_id
WHERE     C.course_id=AW.course_id    AND
          AW.student_id=@student_id
```

D. Watching a Lecture

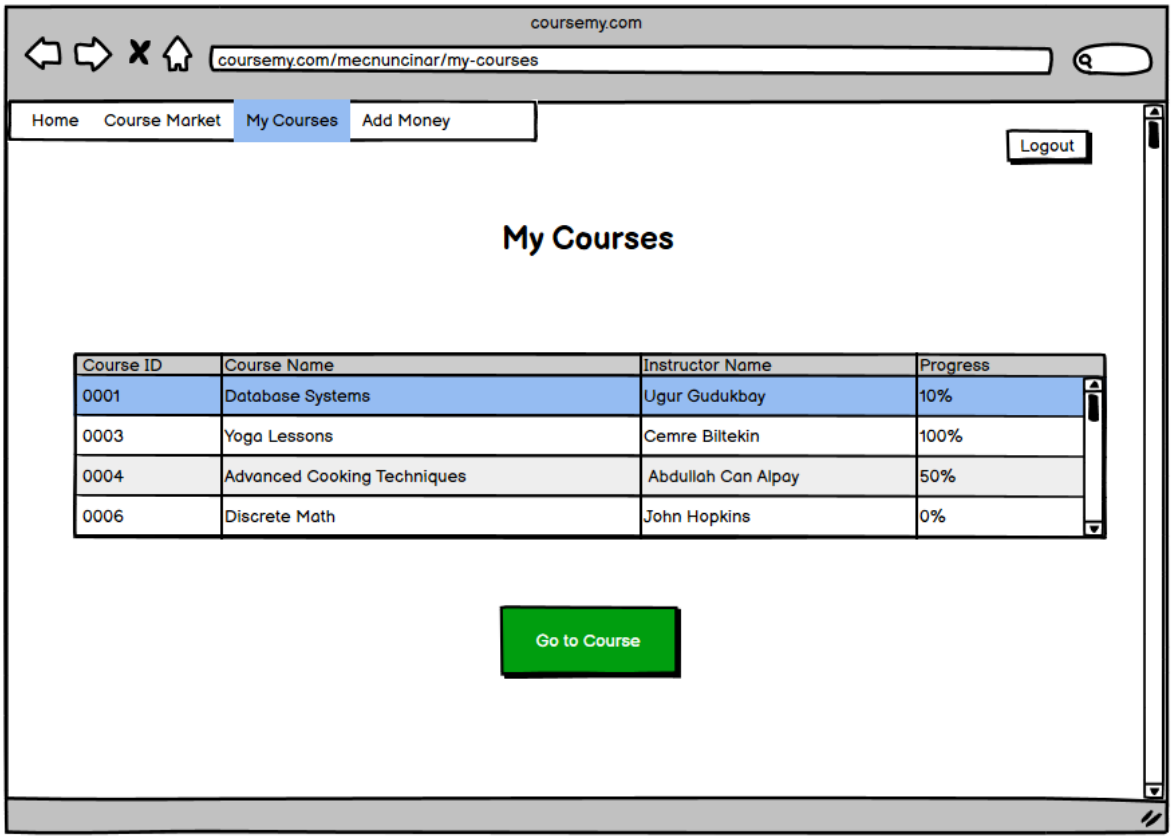

Fig.21: My courses page

First, we retrieve the information about the courses in which the student has enrolled.

SELECT C.course_id, C.course_name, C.average_rating, C.course_description, C.category, E.purchase_date
FROM course C, enrolls E,
WHERE C.course_id = E.course_id **AND** E.student_id = @student_id

In order to calculate the progress of the student for the course, we find the lecture count of the courses.

SELECT course_id, student_id, **COUNT**(lecture_id)
FROM progress
WHERE student_id = @student_id
GROUP BY course_id, student_id

Then, we find the total number of lecture in the course by the following query:
SELECT course_id, **COUNT**(lecture_id)
FROM lecture
GROUP BY course_id

After executing the query above, we get the total number of lectures in the course and the total number of watched lectures in the course. After division, we get the progress and write it onto the table.

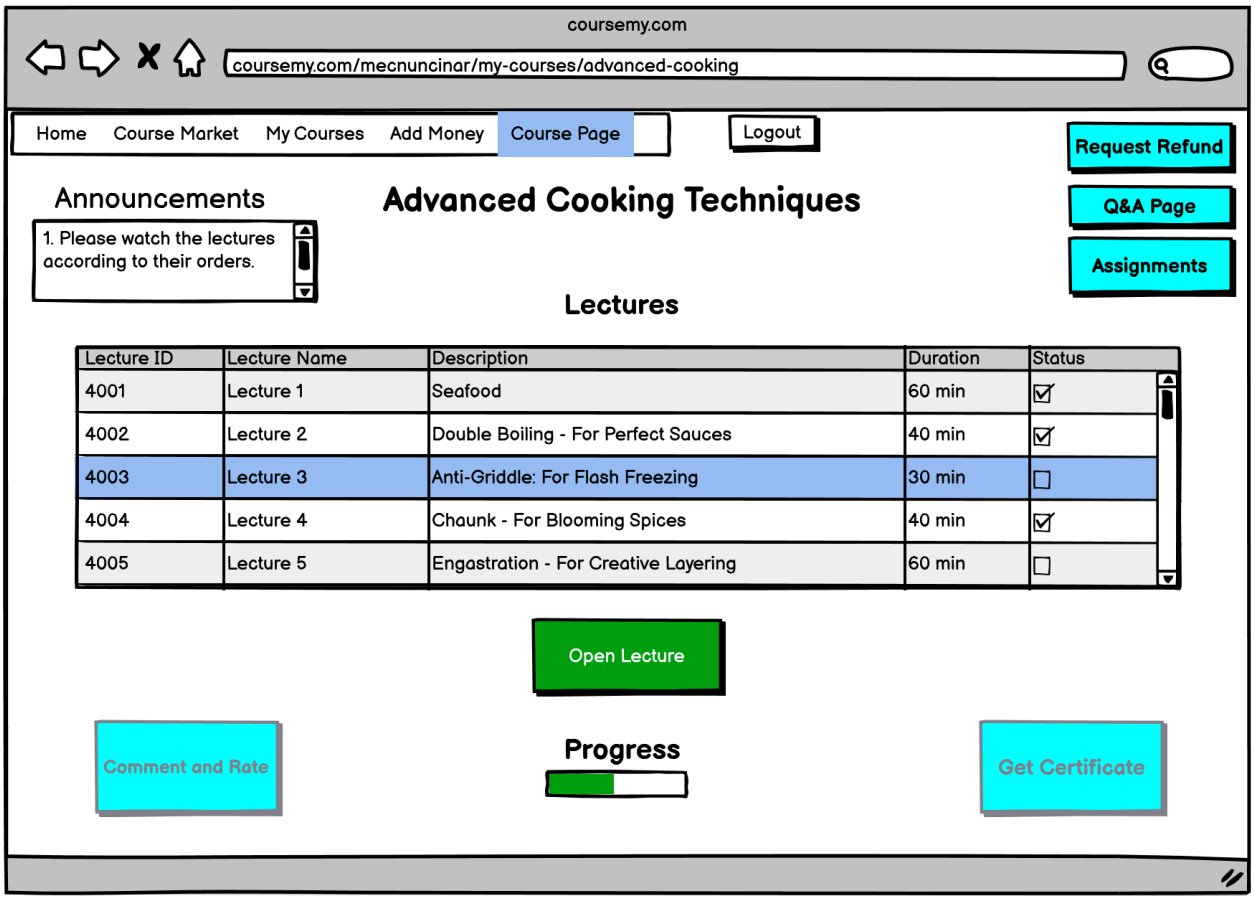

Fig.22: Course page entered from my courses (uncompleted course)

If the progress is not 100%, the student can't comment and rate or get a certificate. Therefore, these buttons are disabled. Finding the progress for the selected course is found as explained at the end of this section.

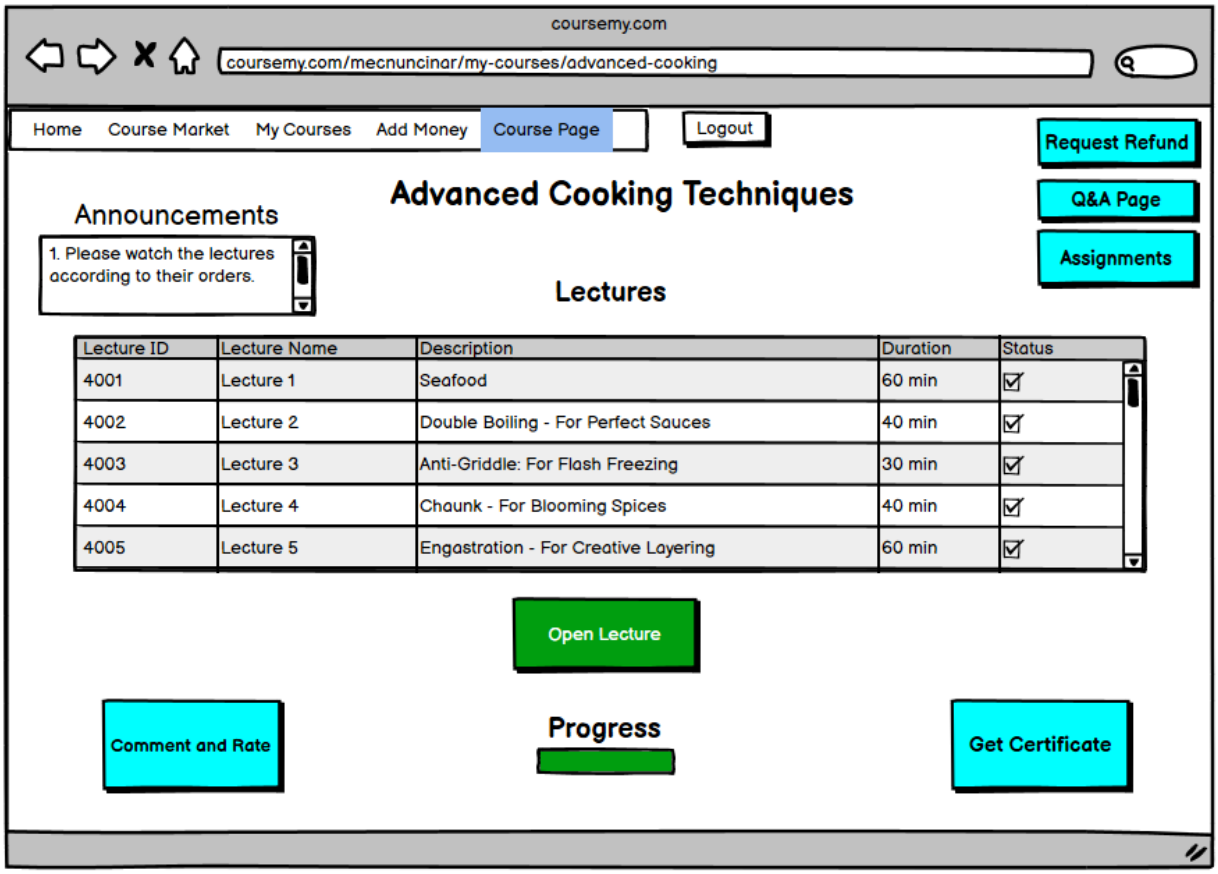

Fig.23: Course page entered from my courses (completed course)

When a course is selected, lectures of that course are displayed to the students.

SQL statement for getting every lecture of a course:

```
SELECT    lecture_name, description, duration
FROM      lecture
WHERE     course_id=@course_id
```

SQL statement for getting every announcement for the course

```
SELECT    text
FROM      announcement
WHERE     course_id = @course_id
ORDER BY  announcement_id DESC
```

The lecture that is selected is opened with the following query.

SQL statement for getting a specific lecture of a course:

```
SELECT    C.course_name, L.lecture_name, L.description, L.duration
FROM      course C, lecture L
WHERE     C.course_id=L.lecture_id
```

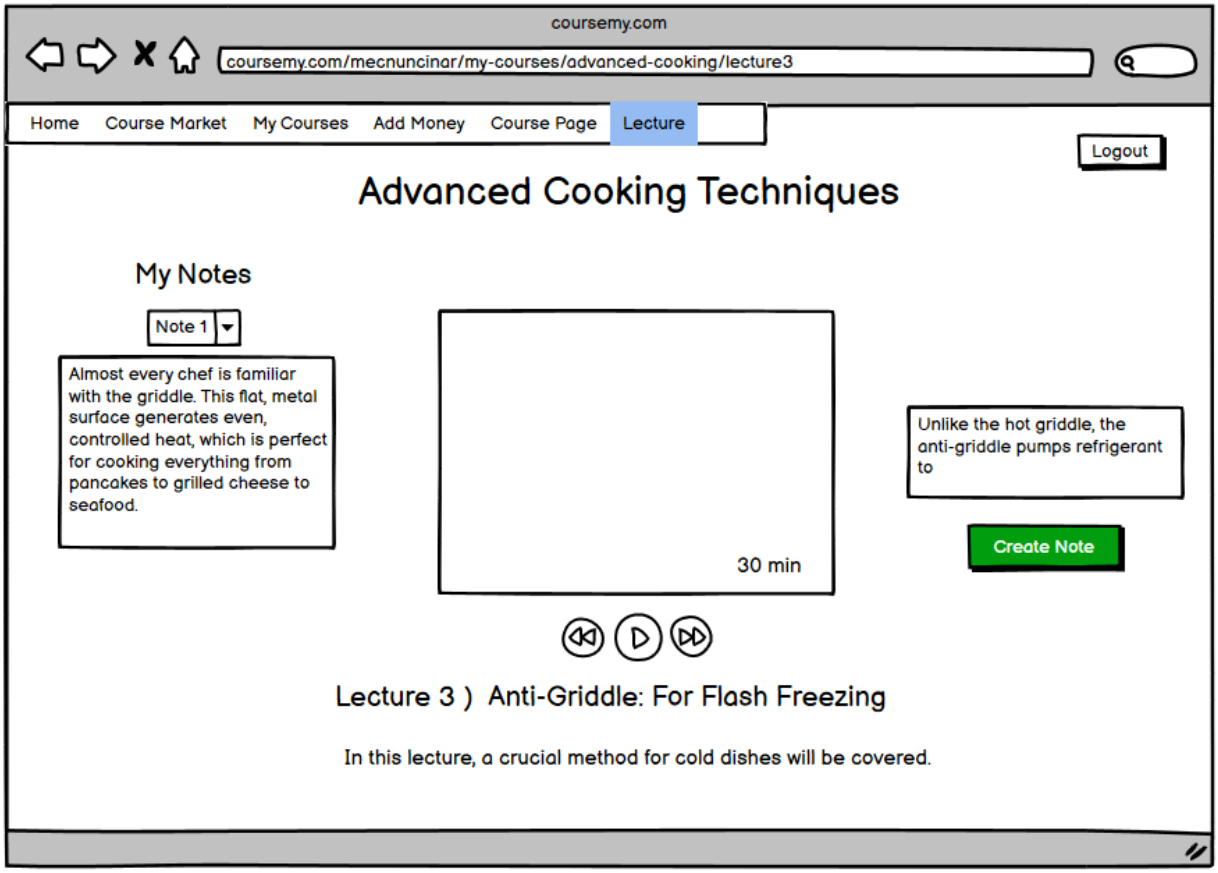

Fig.24: Lecture page of a course

This page is the lecture page where students can watch the lecture. After clicking the play button, the lecture is set to watched, and the lecture of that course is added to the progresses relation of the student. After clicking the next button, the student is directed to the next lecture's page. After clicking the previous button, the student is directed to the previous lecture's page.

The lecture_id's are sorted. I.e., the first lecture has the lowest lecture_id, and the last lecture has the biggest lecture_id.

The id of the current course and the id of the current lecture are stored in @course_id, and @lecture_id respectively.

SQL statement for getting a specific lecture of a course:

```
SELECT    C.course_name, L.lecture_name, L.description, L.duration
FROM      course C, lecture L
WHERE     C.course_id=L.lecture_id
```

If the displayed lecture is the first lecture, then the previous button is disabled, additionally, if the displayed lecture is the last lecture, then the next button is disabled. The lecture is checked if it is the first or the last lecture.

SQL statement to check if the lecture is the first lecture:

```
SELECT    MIN(lecture_id)
FROM      lecture
WHERE     course_id=@course_id
```

The stored @lecture_id and the return value of this query is compared and determined whether the current lecture is the first lecture. If it is, the previous button is disabled.

SQL statement to check if the lecture is the last lecture:

```
SELECT    MAX(lecture_id)
FROM      lecture
WHERE     course_id=@course_id
```

The stored @lecture_id and the return value of this query is compared and determined whether the current lecture is the last lecture. If it is, the next button is disabled.

SQL statement that is executed when next button is clicked:

```
SELECT    lecture_name, duration_description
FROM      lecture
WHERE     course_id=@course_id AND
          lecture_id > @lecture_id
ORDER BY lecture_id ASC
LIMIT 1;
```

SQL statement executed when previous button is clicked:

```
SELECT    lecture_name, duration_description
FROM      lecture
WHERE     course_id=@course_id AND
          lecture_id < @lecture_id
ORDER BY  lecture_id DESC
LIMIT 1;
```

When the lecture is watched, that lecture of that course is added to the progresses relation with @student_id.

SQL statement for inserting the lecture to progresses:

```
INSERT INTO       progresses(student_id, course_id, lecture_id)
VALUES (@student_id, @course_id, @lecture_id);
```

After watching every lecture, and completing all assignments, the student can get a certificate by clicking on the Get Certificate button on the corresponding course page like in *Fig. 23.*

In order to check if a student has watched every lecture in a course, the following queries are executed.

SQL statement for finding the number of lectures that are watched by the student:

```
SELECT    COUNT(lecture_id)
FROM      progresses
WHERE     student_id=@student_id AND course_id=@course_id
```

After executing this query, the following query is executed in order to find the total number of lectures in the course.

```
SELECT    COUNT(lecture_id)
FROM      lecture
WHERE     course_id=@course_id
```

The result of the first and second queries are compared. If they are equal, that means the student has watched every lecture in the course.

In order to determine whether or not the student has completed the course, they also must have completed every assignment. To check,

First, every assignment_id of the course is selected and stored in an array with the following SQL statement:

SQL statement for selecting every assignment of the course:

```
SELECT    assignment_id
FROM      assignment
WHERE     course_id=@course_id
```

We store the length of the array in @num_of_assignment variable in PHP. For every assignment_id in the array, we check if the student has an attempt in which the grade is greater than or equal to the threshold.

```
SELECT    *
FROM      submitted_assignments
WHERE     assignment_id=@assignment_id AND
          student_id=@student_id AND
          grade >= threshold
```

If the following statement returns a nonempty set, we increase the @passed_assignment variable by 1. After doing the same operation for every element in the array, if the @passed_assignment variable is equal to the size of the array, we conclude that the student has completed the course and they are directed to the certificate page.

Fig.25: Certificate page of a course

When a student earns a certificate, a new certificate is created and added to the certificate table. Additionally, information about the student, course and the certificate is added to earn relation. A student can download this certificate.

SQL statement for creating a certificate:

INSERT INTO certificate(date, text)
VALUES (CURRENT_DATE, @certificate_text**)**

Certificate text is generated by PHP with the course name, student name, date, etc.

SQL statement for earning a certificate:
INSERT INTO earns(student_id, course_id, certificate_id)
VALUES (@student_id, @course_id, @certificate_id**)**

E. Adding a Note to a Lecture

A student can add a note to a lecture for themselves as can be seen from *Fig. 24*. The notes created by a student are only visible to the students who created the notes. @note_text contains the note that is entered by the student. When the student enters a text to the note text field and clicks the create note button, the following query is executed.

SQL statement for creating a note:

```
INSERT INTO      note(student_id, lecture_id, course_id, note_text)
VALUES (@student_id, @lecture_id, @course_id, @note_text)
```

When the student starts watching a lecture, the following query is executed to get all notes of the student of a lecture. The result is stored in an array and shown accordingly. For instance, if Note 1 is selected, the first element of the array is shown.

SQL statement for getting all notes of a student for a lecture:

```
SELECT      lecture_id, note_text
FROM        note
WHERE       student_id=@student_id AND
            lecture_id=@lecture_id  AND
            course_id = @course_id
```

F. Feedback for a Course

If a student finishes the course and gets their certificate, they can go to comment/rate page by clicking the Send Feedback button.

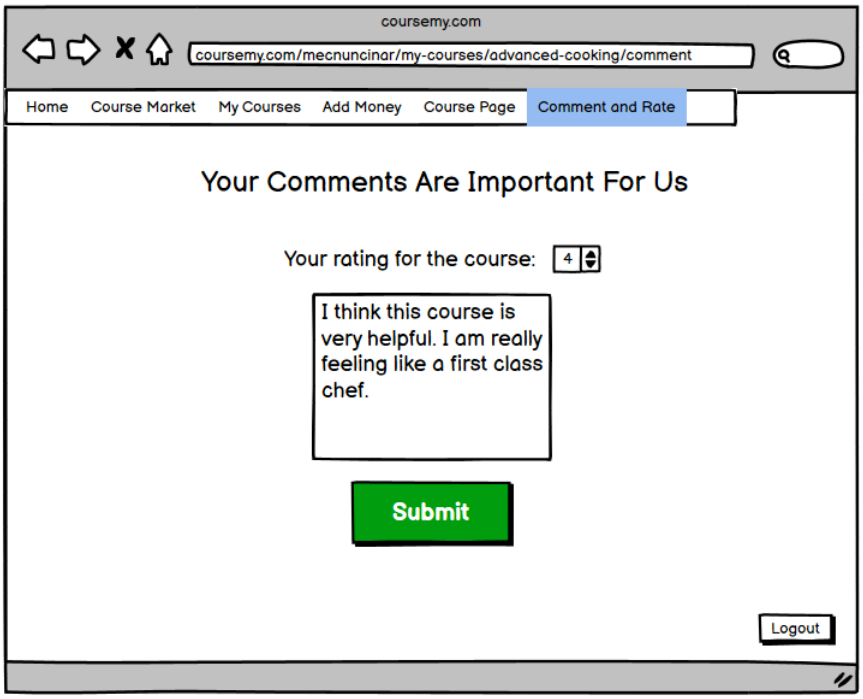

Fig.26: Feedback creation page

In that page, the students can comment on and rate the course that they have finished. First of all, the feedback is created as follows.

SQL statement for creating a feedback:

```
INSERT INTO feedback(feedback_note, rating)
VALUES (@entered_note, @entered_rating)
```

The id of the added feedback is stored in @feedback_id.

SQL statement for adding feedback to the student feedbacks:

```
INSERT INTO student_feedbacks(student_id, course_id, feedback_id)
VALUES (@student_id, @course_id, @feedback_id)
```

II. Publish a Course by Course Creator

A course creator can create a new course by clicking the Create a New Course button on the navigation bar.

A. Specify Course Features Page

The course creator accesses this page to create a new course. This page asks for features of the course to be entered by the course creator, adding lectures and assignments and their features. A course is not added to the database until all necessary information about a course is determined, course creator is satisfied with the lectures and assignments he/she added, and course creator presses the Publish Course button lastly.

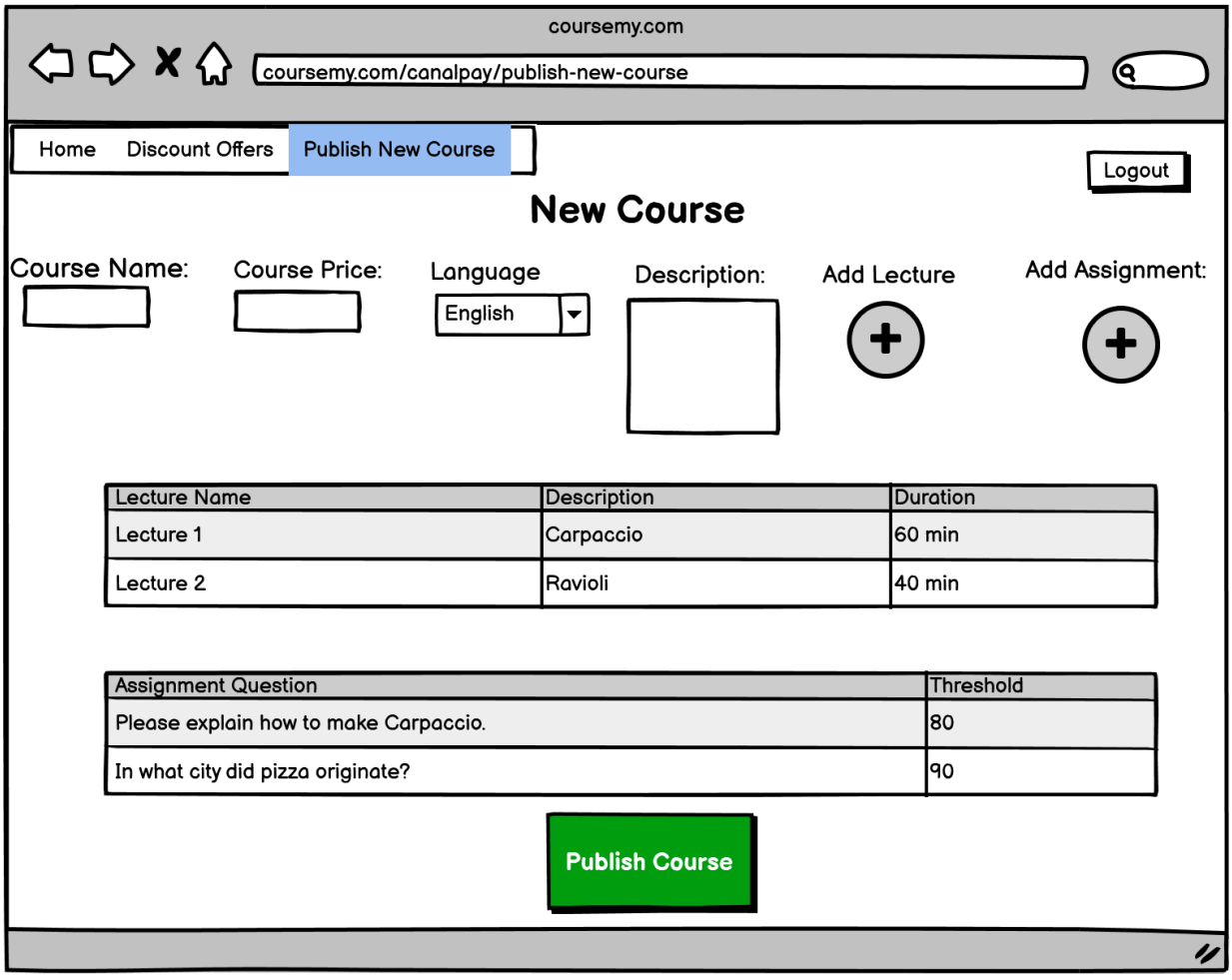

Fig. 27: Course creation page for course creator use

After clicking on the Publish button, the following queries are executed:

SQL statement for adding a new course by a course creator:

```
INSERT INTO     course(course_name, language,
                    course_price, create_date, average_rating,
                    category, course_description, certificate_price,
                    course_creator_id)
VALUES      (@course_name, @language, @course_price, CURRENT_DATE, 0,
            @category, @course_description, @certificate_price,
            @course_creator_id);
```

The above query adds a new course by the course instructor whose id (@course_creator_id) is stored in the website to the database upon login. Course's id is automatically generated by the database. If the course_creator does not specify features that cannot be null, the page gives a warning about spaces that cannot be left blank, and creation is unsuccessful in this case.

B. Create a Lecture For A Course Page

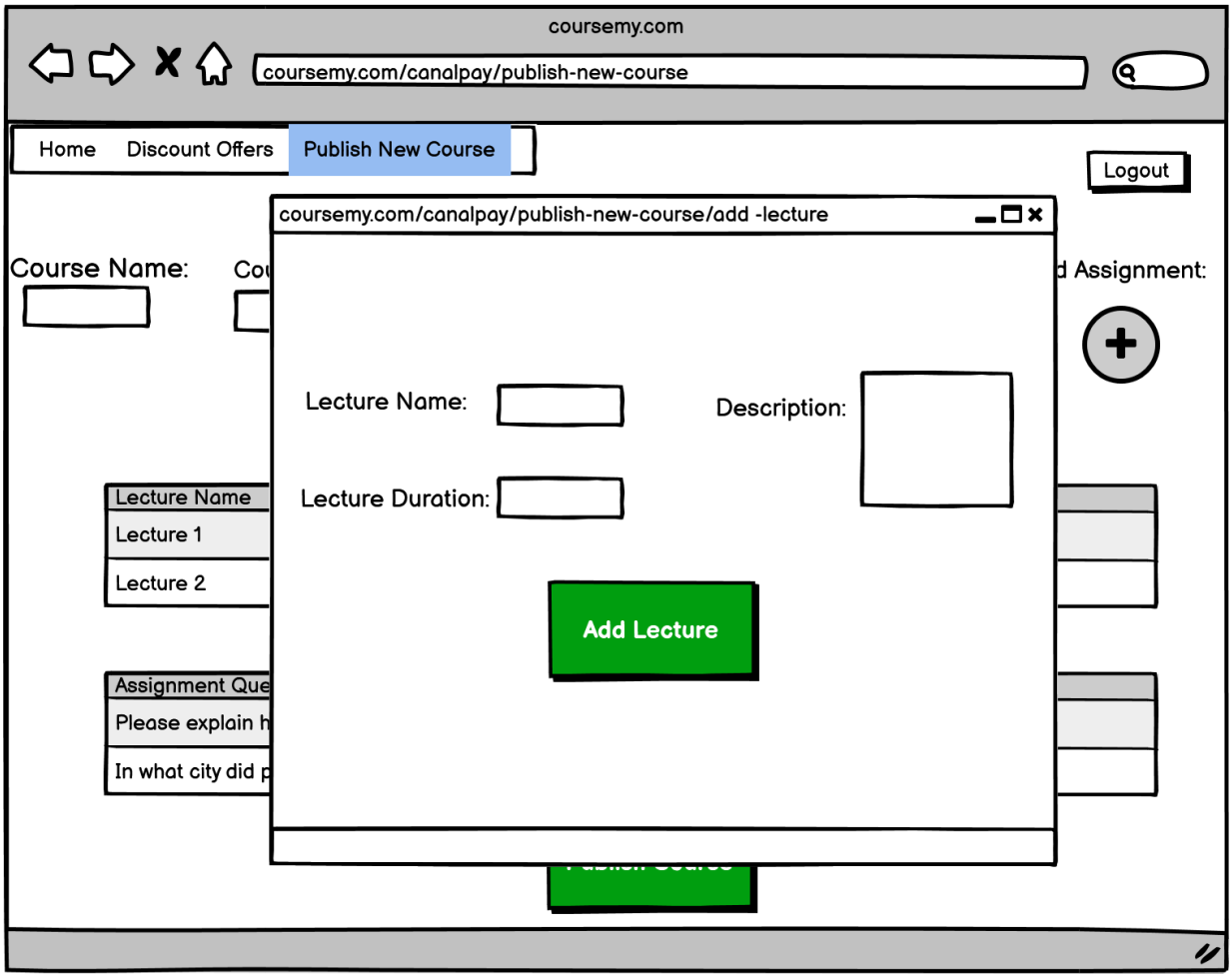

Fig. 28: Lecture creation window by clicking on Add Lecture button for course creator

The course creator can choose to add a new lecture to a course in the creation process. The page asks for features for the new lecture to be added. After specifying the lectures, the course creator can publish the course with these lectures.

First, after SQL statement for course creation is executed, the newly generated course's id is taken for lecture addition to the database purpose like the following:

SET @course_id = SCOPE_IDENTITY()

SQL statement for adding a new lecture to course:

INSERT INTO lecture(course_id, lecture_name, duration, description)
VALUES (@course_id, @lecture_name, @duration, @description)

Also, the course creator has to specify assignments and their features like he/she does with lecture features and addition. The course creator clicks on the Add Assignment button to add an assignment to the course in the creation process. After the Publish button is clicked and the course is inserted to the database with a unique id, similarly to lectures, assignments are also added to the database.

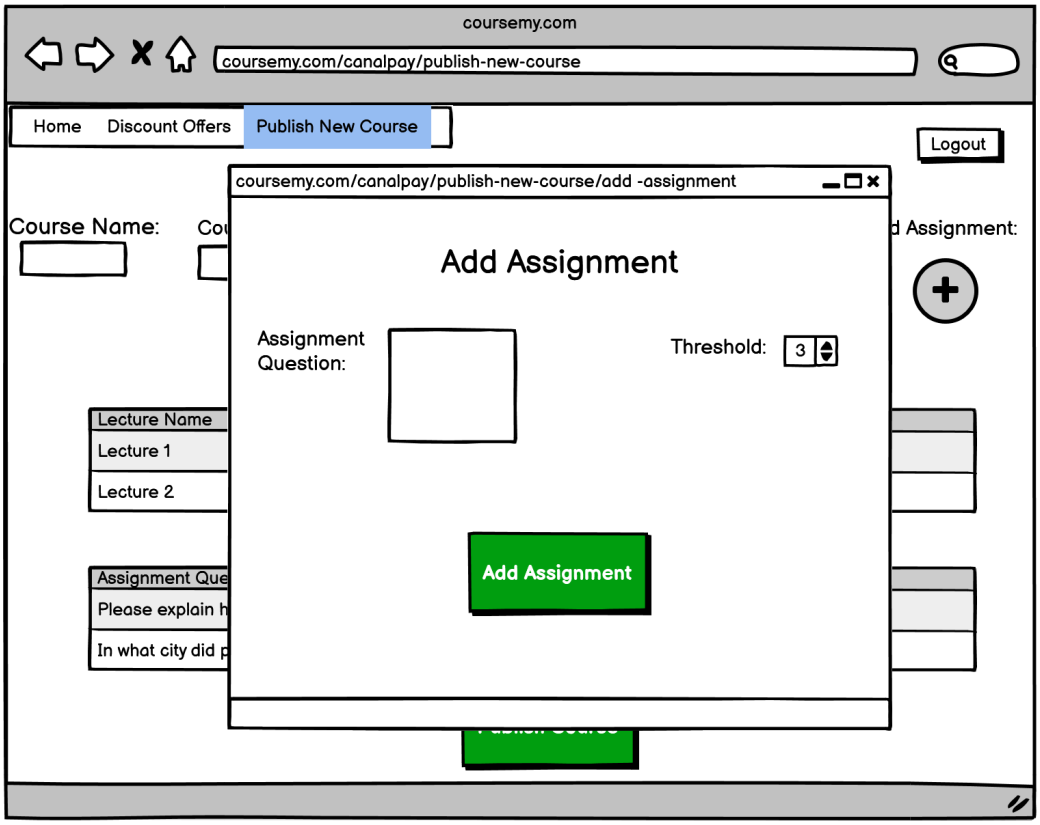

Fig. 29: Assignment creation page for course creator

SQL statement for adding a new assignment to the course:

```
INSERT INTO      assignment( assignment_question, threshold, course_id)
VALUES  (@question, @threshold, @course_id)
```

In the above query, questions and thresholds are read from text fields as input. Course id is stored in software as above.

C. Make an Announcement for a Course Page

First, the course creator has to select a course to make an announcement on. The course creator can view all his/her offered courses from My Courses page that can be viewed by a course creator user like in *Fig.7*.

SQL statement to show all offered courses of a course creator:

```
SELECT    C.course_id, C.course_name, C.average_rating, total_student,
          D.percentage, (CASE WHEN CURRENT_DATE >= D.start_date AND
                         CURRENT_DATE < D.end_date AND D.is_allowed
                         THEN C.course_price * (( 100 - D.percentage ) / 100)
                         ELSE c.course_price END) as price
FROM      course C LEFT OUTER JOIN discount D ON
          C.course_id = D.discounted_course_id,
          (SELECT    course_id, COUNT(student_id) AS total_student
          FROM       enrolls
          WHERE      course_id IN (SELECT course_id
                                   FROM course
                                   WHERE course_creator_id = @course_creator_id)
          GROUP BY   course_id) course_student
WHERE     course_student.course_id = C.course_id
```

For the above statement, @course_creator_id is the current course creator's id that is stored on the session and using this web application.

Then, the course_creator can choose one of his/her offered courses from this list to go to that course's page. Let @course_id be the fetched course id upon choosing one of the courses and its value is carried over to the my courses page.

SQL statement to retrieve the selected course's name from the list:

```
SELECT    C.course_name
FROM      course C
WHERE     C.course_id = @course_id
```

SQL statement to retrieve the lectures of the selected course:

```
SELECT    L.lecture_name, L.description, L.duration
FROM      lecture L
WHERE     L.course_id = @course_id
```

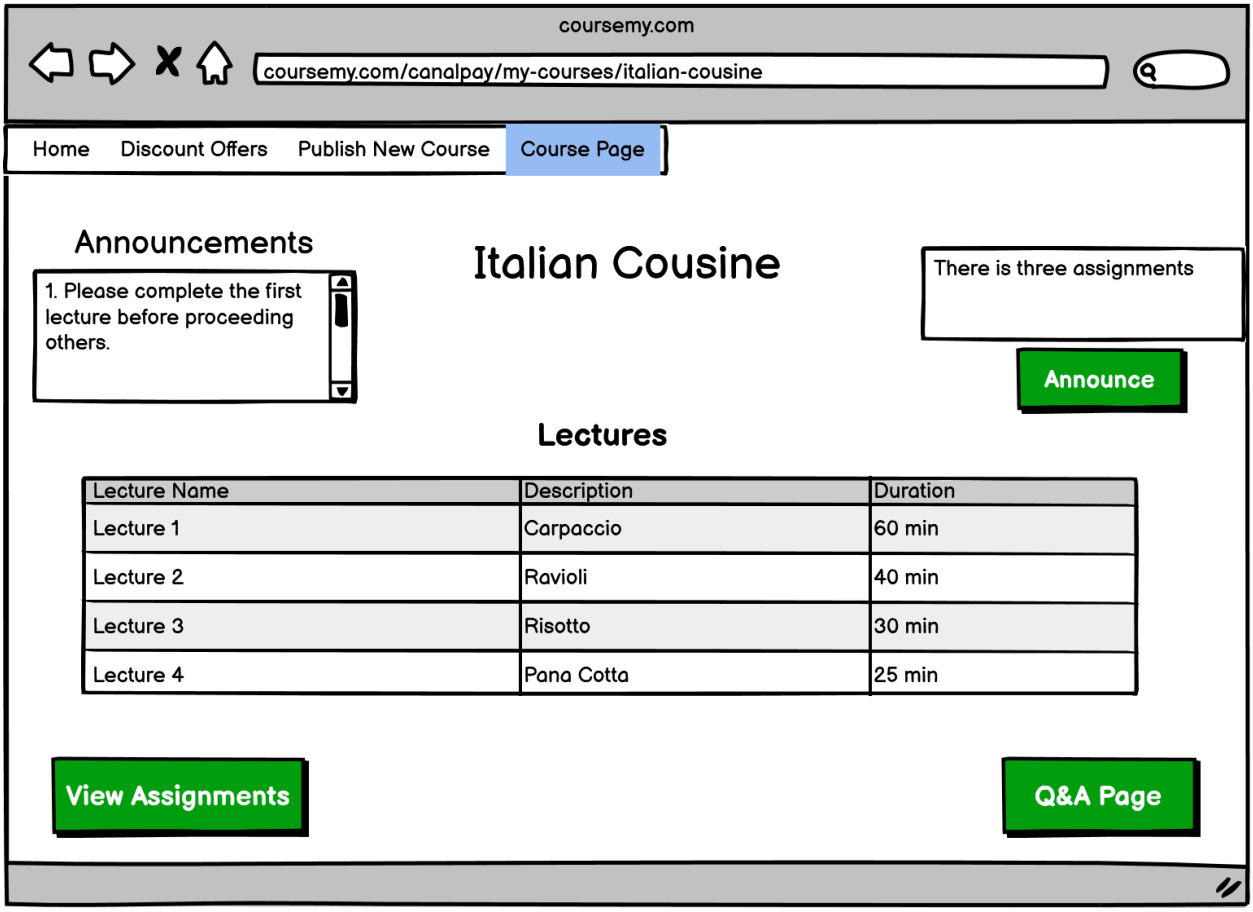

Fig. 30: Course page from course creator's perspective

A course creator can make an announcement by choosing one of his/her courses and going to the course's page. To create an announcement, the course creator must fill the announcement text field and then press the Announce button. List of announcements can be browsed from the Announcements feed on the page. If the Announce button is pushed with empty text input, then an error message is generated and making an announcement is unsuccessful.

SQL statement for creating a new announcement for a course:

```
INSERT INTO       announcement(text, date, course_id)
VALUES (@text, CURRENT_DATE, @course_id)
```

For the above statement, @course_id is carried to the course page as stated previously and is the displayed course's course id on this page.

SQL statement for displaying all announcements for a course in creation order:

```
SELECT      text
FROM        announcement
WHERE       course_id = @course_id
ORDER BY    announcement_id DESC
```

The above statement finds all announcements of a course, orders them by creation order, and these queried announcement texts will be stored in an array to display them with indices on this page as can be seen from the figure.

Creation of Announcement Notifications:

We know the announcement is added to the @course_id. When a new announcement is added to the entity, we know which students to send the notification, the ones that are taking the course. Thus, to find this student list we need the following SQL.

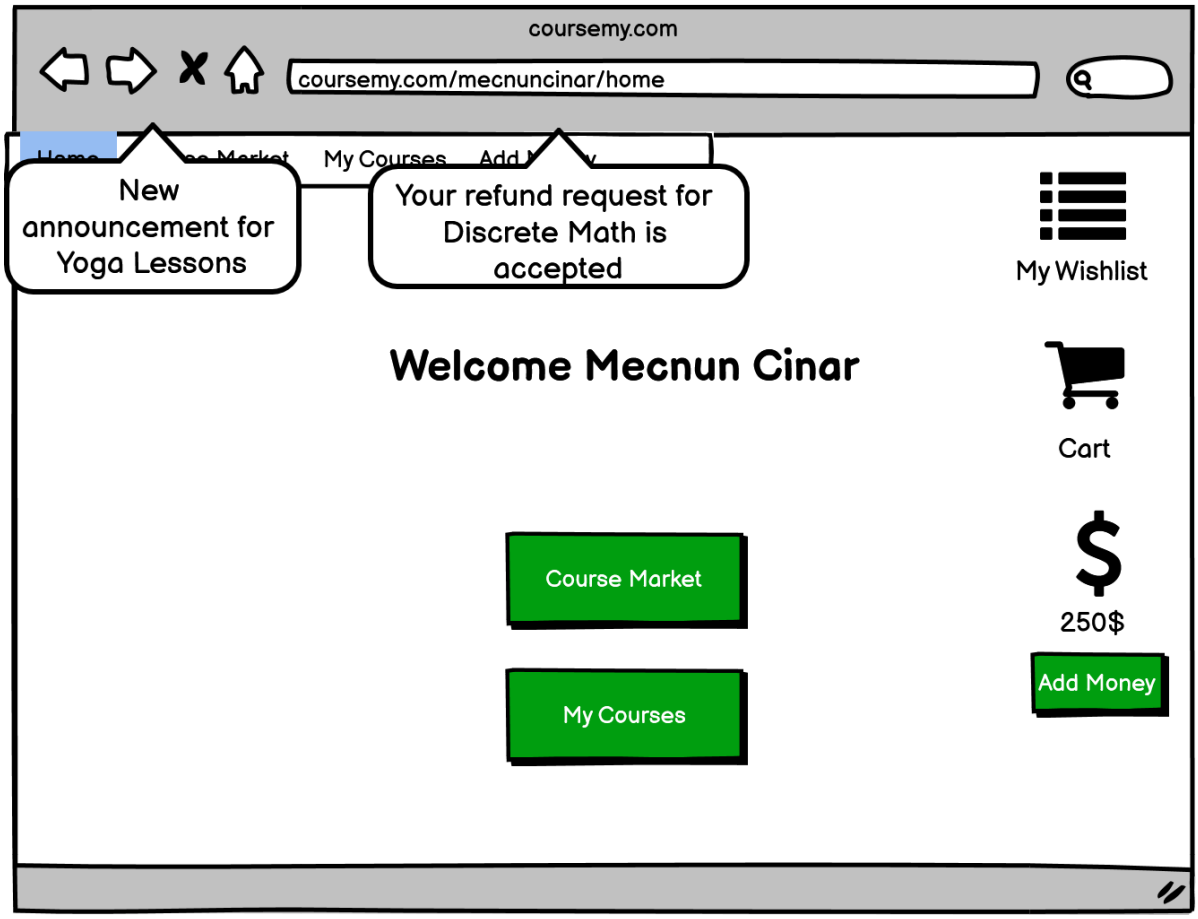

Fig. 31: Notification screen

SQL statement to find students taking a particular course:

```
SELECT    student_id
FROM      enrolls
WHERE     course_id = @course_id
```

After that when we have the student id's, we can simply use the php library pushpad\Notification to notify the relevant students.

D. Course Q&A Page

Select a course and/or lecture to ask a question about (by a standard user)

When a student selects a course and clicks to 'ask question', we will have @student_id (which is the current user) and @course_id (which is retrieved from php when the student selected the course). Thus, we only need the question text. When the student enters the question text the php will store it as @question_text.

In php, we can retrieve the id of the question via MYSQLi->inserted_id. In the following question id is referred as @question_id.

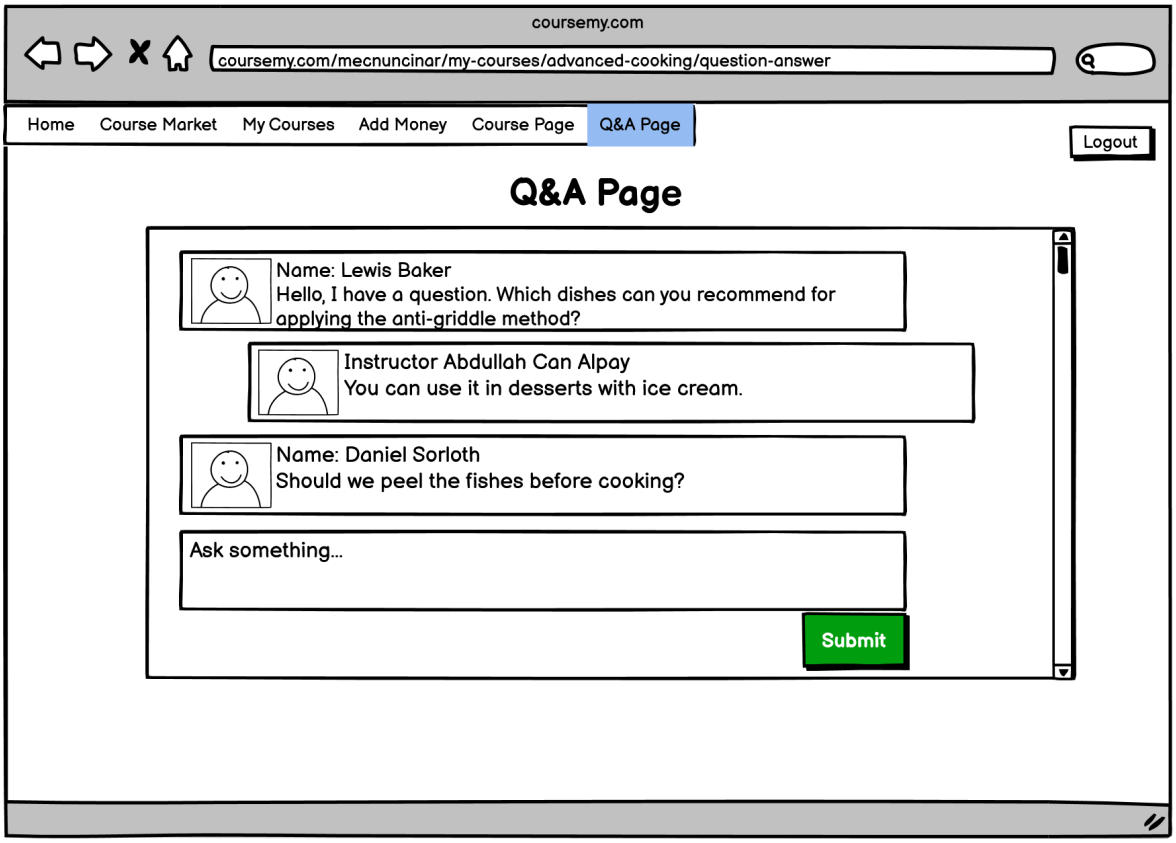

Fig. 32: Q&A page of a course from students' perspective

SQL statements to ask a question:

INSERT INTO question **VALUES** (**CURRENT_DATE**, @question_text)

Then, we need to specify which student asks this question for which course.

INSERT INTO asks **VALUES**(@question_id, @student_id, @course_id)

With that we will store the questions.

Course creator lists its courses and selects one:

In the first query, all the courses created by the current course creator (@course_creator_id will store the user's id, which is the course creator). In the second query, the course creator selects a course and all the information is displayed for the course creator.
@course_id refers to the selected course.

SQL statements to display created courses and retrieve the selected course information:

```
SELECT    C.course_id, C.course_name, C.course_rating, C.create_date,
          C.category
FROM      courses C
WHERE     C.course_creator_id = @course_creator_id
```

```
SELECT    *
FROM      course C
WHERE     C.course_id = @course_id
```

List questions about that course

We still have the @course_id in php.

SQL statement to display questions of specific course:

```
SELECT    Q.question_id, Q.question_text
FROM      asks A, question Q
WHERE     A.courses_id = @course_id AND A.question_id = Q.question_id
          AND A.question_id NOT IN (SELECT    Ans.question_id
                                    FROM      answers Ans)
```

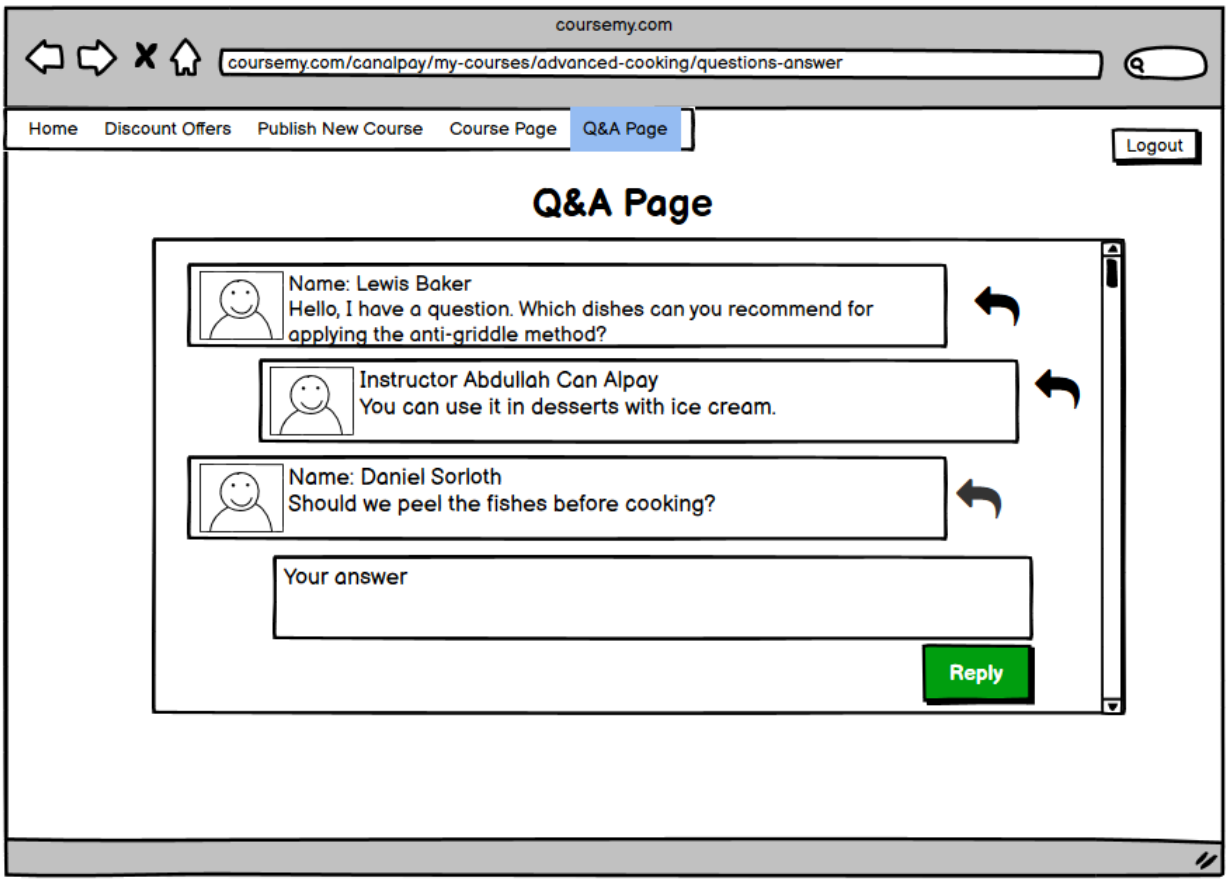

Fig. 33: Q&A page of a course from course creators" perspective

Now, we display the questions and we have their id's.

Select one question and answer it

When course_creator selects one of the questions, we can access it since we know the question's id from the php. @question_id refers to the selected question.

When the answer text is filled, we can insert into the answers entity and we can access the answer text from php which is referred as @answer_text.

SQL statement to answer a question:

INSERT INTO answers **VALUES** (@question_id, @answer_text)

III. Site management (by an Admin)

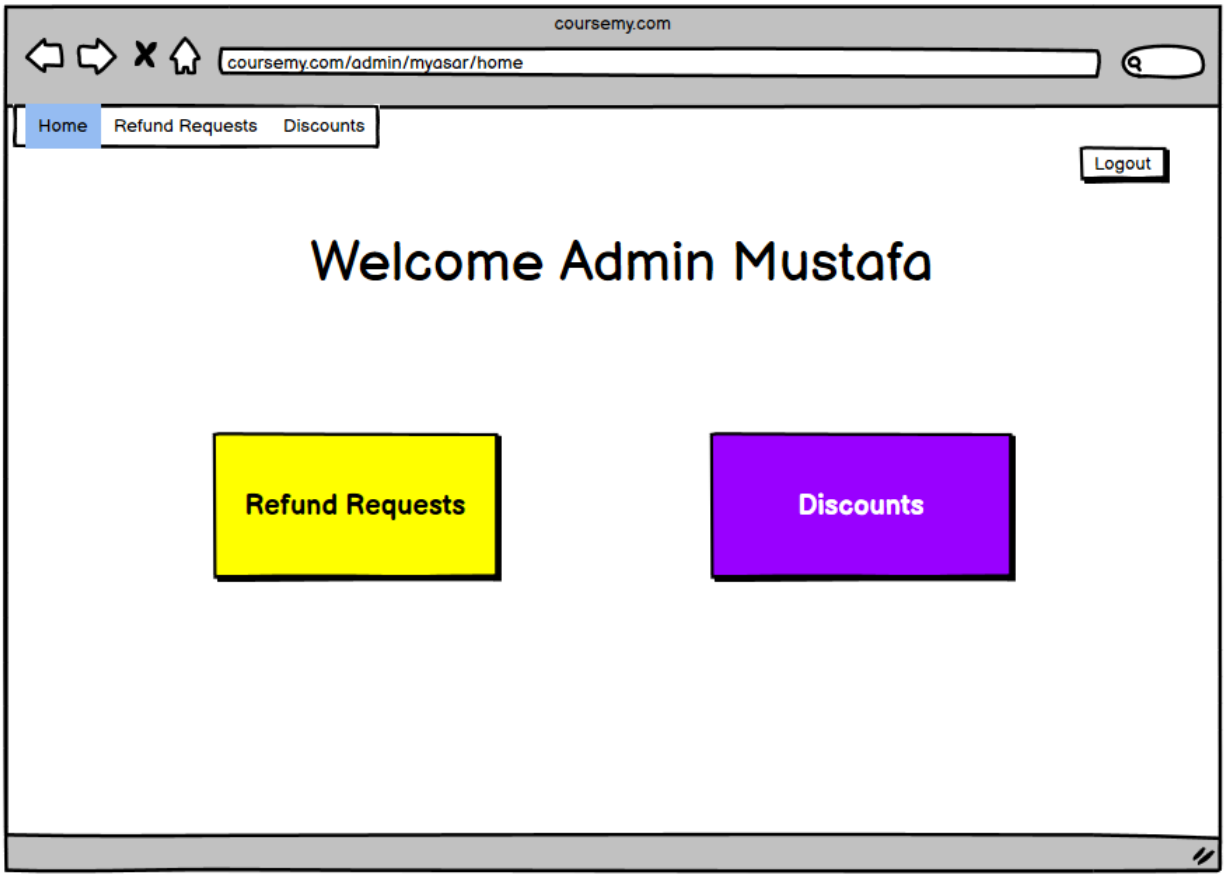

Fig. 34: Home page of site admin

As an admin, s/he is responsible for maintenance, addressing refund requests and offering discounts. When a student sends a refund request, the admin's refund page will be updated and new requests will be added.

A. Request refund on a course (by a standard user)

Students can make refund requests from their course list by selecting the course and clicking the ‘Request Refund’ button. Then in the opened page, students will write the refund reason.

List all bought courses

From the below SQL, the courses bought by the student will be displayed. @student_id comes from the php, it refers to the current user’s id.

SQL statement for displaying the courses:

SELECT C.course_id, C.course_name, C.language, C.average_rating, C.category, C.course_description, E.purchase_date
FROM course C, enrolls E,
WHERE C.course_id = E.course_id **AND** E.student_id=@student_id

Select the course to return

When we display the enrolled courses for the student, we also get the course_id's of these courses and stored in the php. When the student clicks the course, from php we will know which course is selected and stored in @course_id.

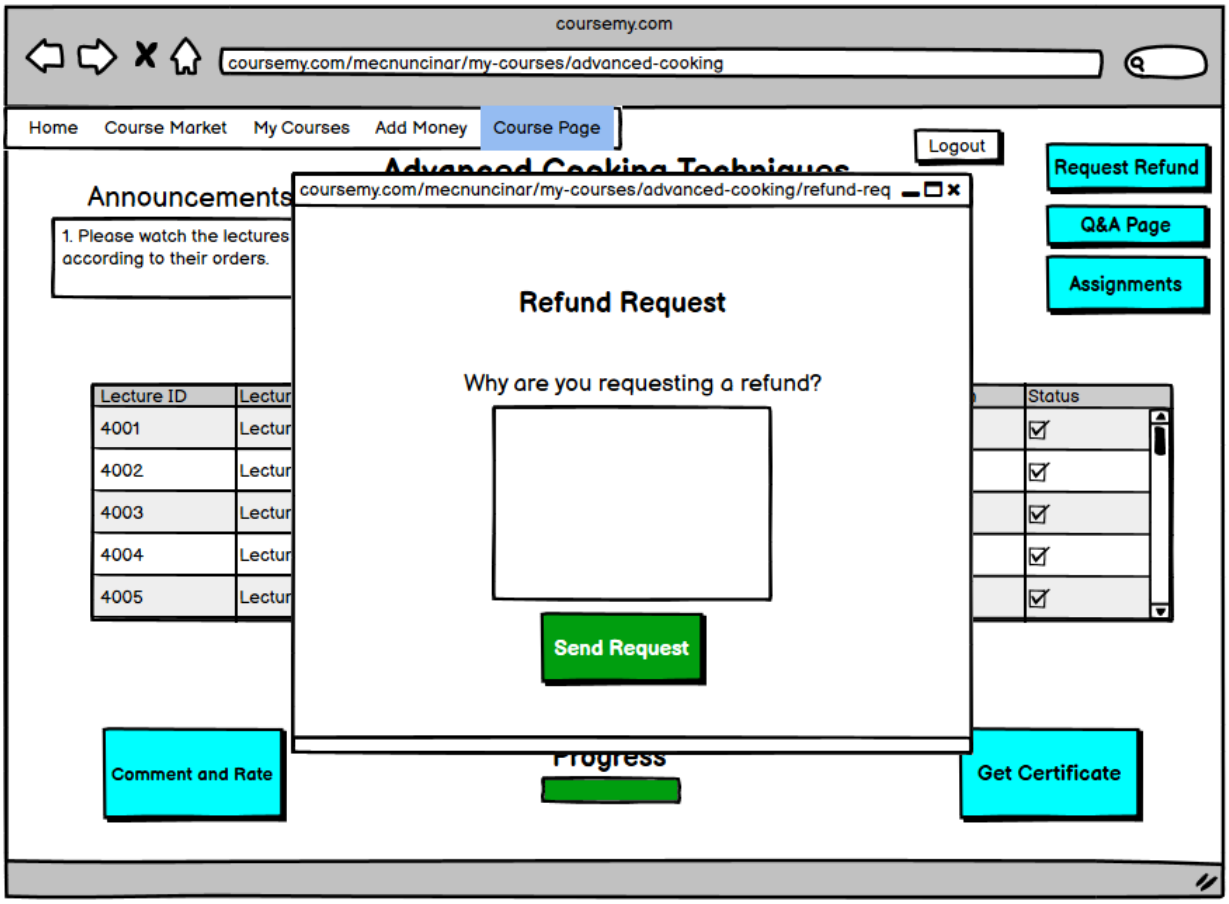

Fig. 35: Refund screen for student

Specify the reason for the request and send

When the student writes the reason for the refund, we will create a new refund with the @refund_reason and refund_id. Refund_ids are auto incremented attributes, thus, we do not need to specify while inserting into the refund entity. Thus, we can simply add the refunds in the refund entity.

After inserting refund requests into the refund entity, we need to specify the course_id and the student_id. We also need to add this tuple into the refund_requests entity. So that, when the admin responds to the refund request, we can update the student's course.

As it mentioned before we do not store the refund_id yet. We know that this refund is the lastly added refund, thus, with the MySQLi->inserted_id will return the id of lastly added refund. We will refer to it as @refund_id.

SQL statements after student requests a refund:

INSERT INTO refund(reason) **VALUES**(@refund_reason)

And we also insert the refund request in the refund_requests table.

INSERT INTO refund_requests **VALUES**(@refund_id, @student_id, @course_id)

B. Check request and approve/reject (by an admin)

Firstly, we will display the refund requests. We do not need to show previously assessed requests for admin. When the admin selects a refund request and sends a response, we need to inform the relevant student.

List available refund requests

We will display not previously assessed refunds.

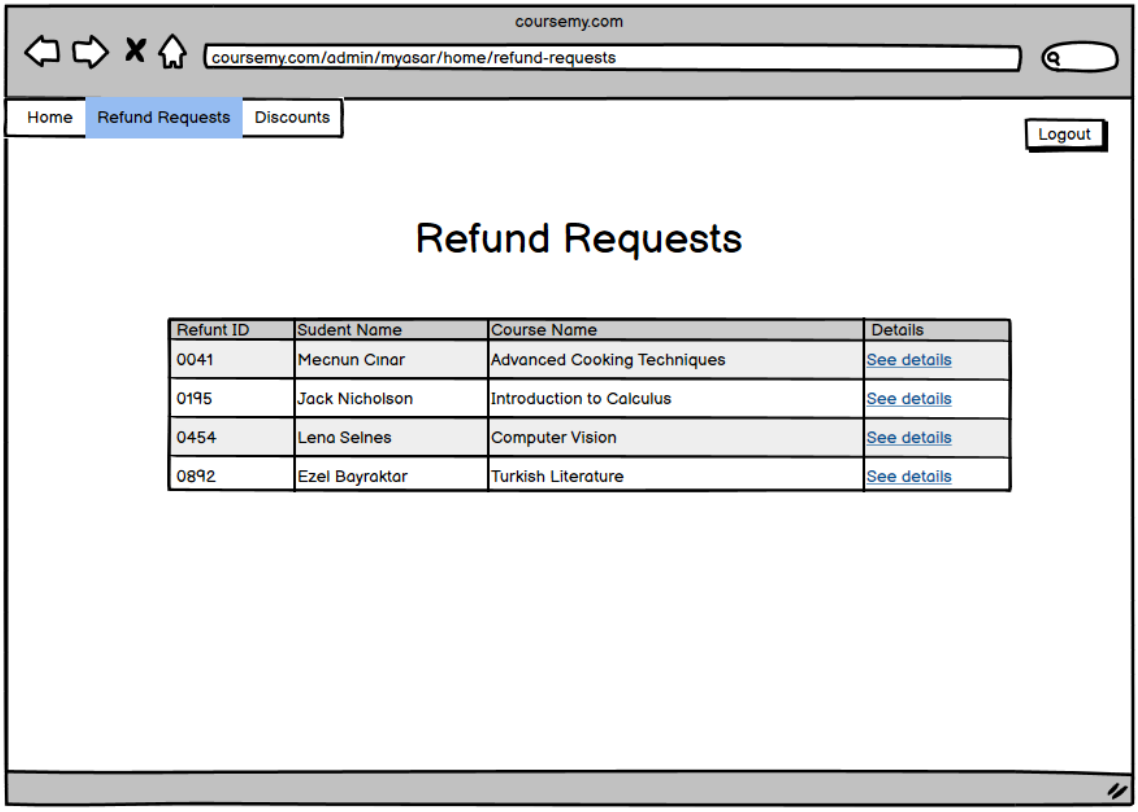

Fig. 36: List of refund requests from admin's perspective

SQL statement for displaying refunds:

```
SELECT    R.refund_id,  R.reason, RR.course_id, RR.student_id
FROM      refund R, refund_requests RR
WHERE     R.is_assessed = 0 AND RR.refund_id = R.refund_id
```

Select a refund request and make a decision

Previously, when we displayed the refund requests, we also retrieved the refund_ids. When the admin selects a request, PHP will store refund_id in @refund_id. After the admin assesses the request, is_assesed will be TRUE and the outcome will be stored in is_approved.

@selection comes from the php. Admin will select the ‘Approve’ or ‘Deny’ option according to reason and purchase date. If the admin selects the ‘Approve’ option, @selection will be TRUE vice versa otherwise.

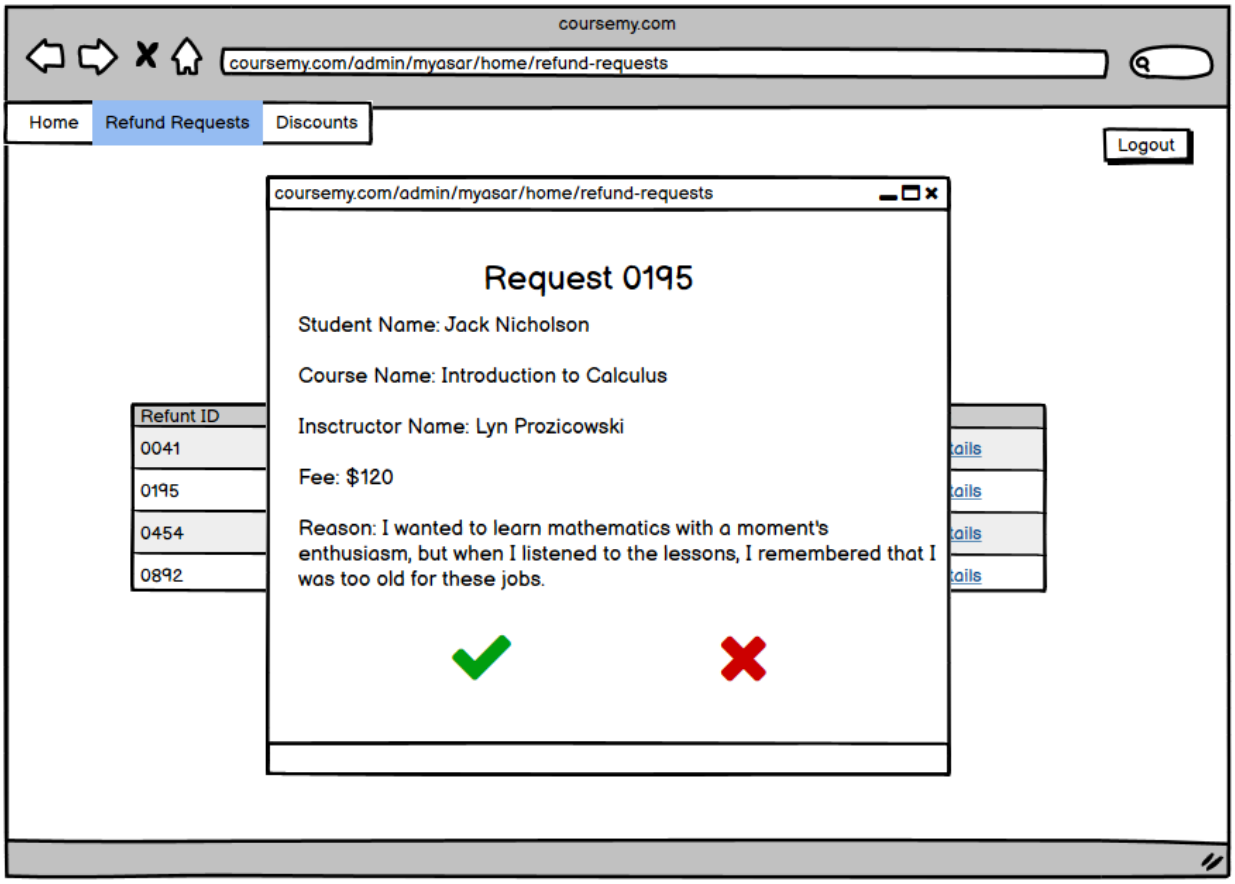

Fig. 37: Details of a refund request

SQL statement for assessing refunds:

UPDATE refund
SET is_assessed = TRUE, is_approved = @selection
WHERE refund_id **IN** @refund_id

User must be notified of outcome of request

When the is_assessed attribute becomes true, we will know that the refund is assessed. From that we will have following conditions:

If is_assessed is false => Refund is not addressed yet.
If is_assessed is true and is_approved is false => Refund is assessed and rejected.
If is_assessed is true and is_approved is true => Refund is assessed and accepted.

When the refund is accepted (is_assessed and is_approved returns true), the course will be deleted from the student's enrollments and the price will be added to the student's wallet.

We get the refund_id, student_id, course_id and is_approved from the first SQL and stored in php as @refund_id, @student_id, @course_id, @result, respectively. When the admin responds to the refund, the refund entity will be updated and then we get the above attributes for only one refund. We delete the assessed refund and when the admin addresses one more refund, we also retrieve that information too.

We get the assessed refunds, both accepted and rejected.

@student_id, @course_id, @result are an array; where i hold the i-th assessed refund information.

SQL statements for addressing refunds:

```
SELECT      R.refund_id, RR.student_id, RR.course_id, R.is_approved
FROM        refund R, refund_requests = RR
WHERE       R.refund_id = RR.refund_id AND R.is_assessed = TRUE
```

When we retrieve the necessary information, we can simply notify the relevant student via php Library pushpad\Notification. Notification UI can be seen in figure 31.

When we notify the student we do not need to hold the refund request in the dataset. So we simply delete it from the set.

```
DELETE FROM     refund_requests
WHERE           refund_id IN @refund_id
```

If the @result is TRUE we delete the course from the student's enrollments and add the price to the wallet.

```
@price =
SELECT      E.purchased_price
FROM        enrolls E
WHERE       E.course_id = @course_id AND E.student_id = @student_id

UPDATE      student
SET         wallet = @price + wallet
WHERE       student_id = @student_id

DELETE FROM     enrolls
WHERE           student_id = @student_id AND course_id = @course_id
```

C. Offer discount for a course

The admin can offer discounts for courses. The discounts will be evaluated by the course creator and according to the course creator's response, the discount will become effective or not. The discounts will remain effective until the end date of the discount. Additionally, while one discount is effective it can be cancelled by the admin before the end date.

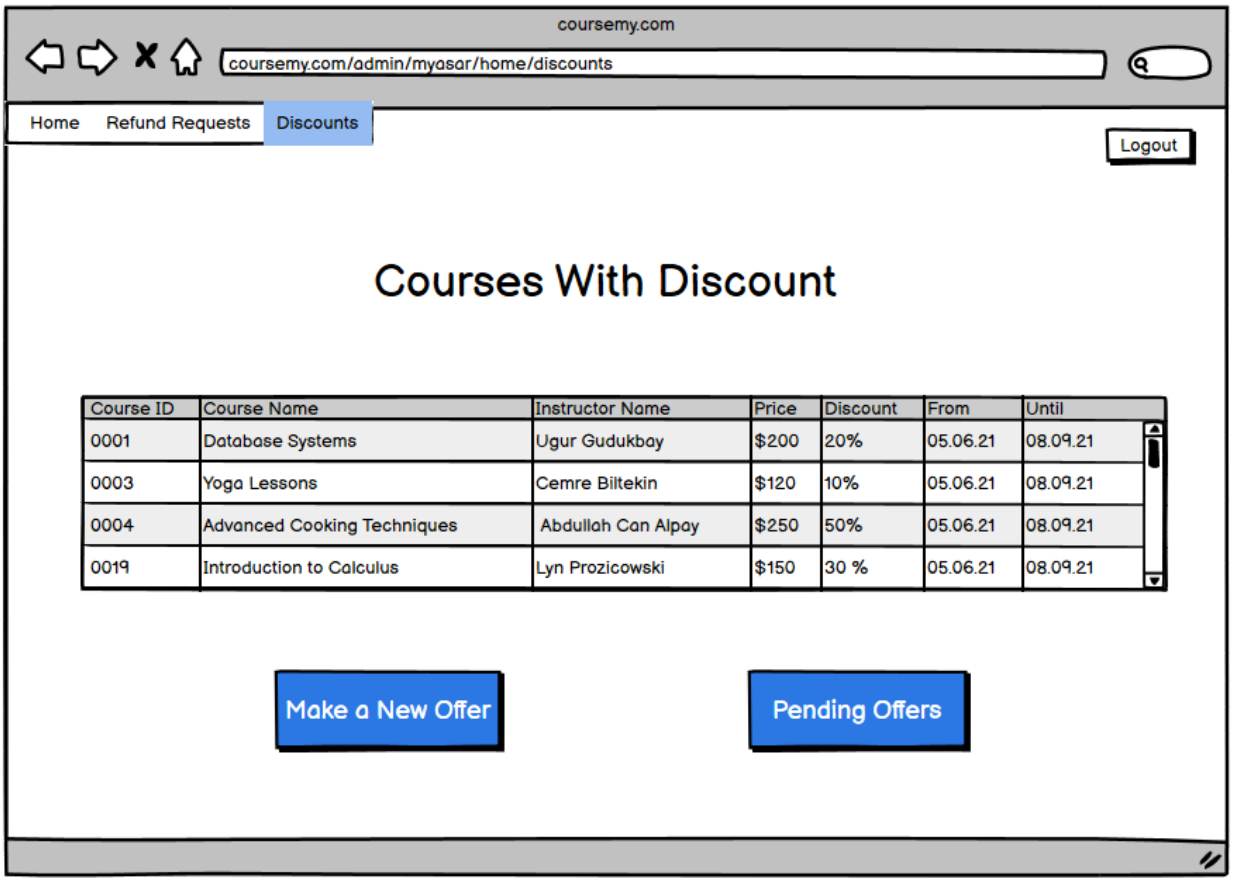

Fig. 38: List of discount allowed courses.

List all discount allowed courses

When one discount is effective for a course, the course cannot take any additional discount. Thus, these courses will not be displayed at the offer discount list.

SQL statement for displaying discount allowed courses:

```
SELECT    C.course_id, C.course_name
FROM      course C
WHERE     C.course_id NOT IN (SELECT    D.discounted_course_id,
                              FROM      discount D
                              WHERE     D.is_allowed = 1 AND
                                        D.end_date > CURRENT_DATE)
```

Select a course and apply desired discount

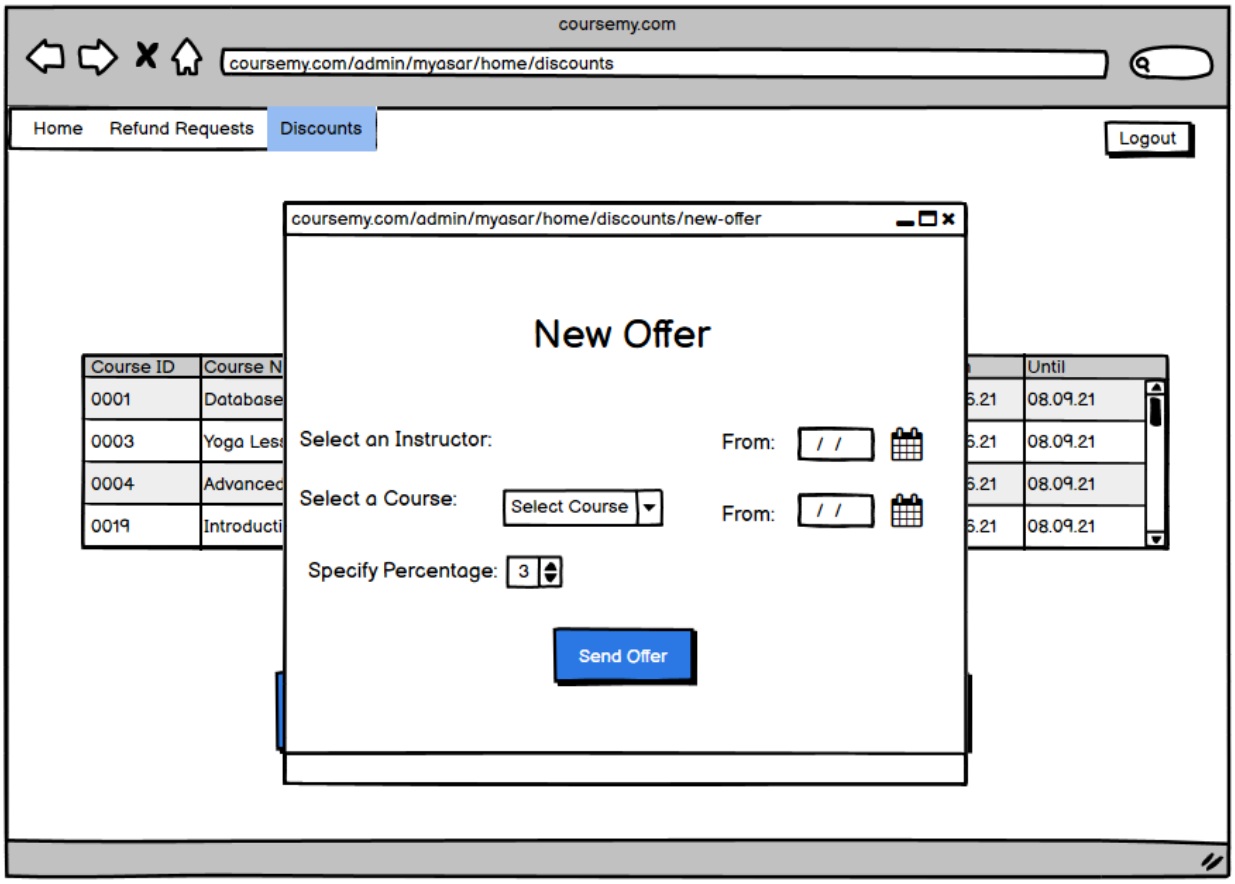

Fig. 39: Discount offer sending screen

To apply the desired discount, the admin needs to make an offer for a course and the course creator should accept this offer. After that, we can apply the desired discount for the course.

Firstly, the admin needs to make a discount offer for the selected course. We stored all the displayed courses' id in php, and when the admin selects a course, we will have the course's id as @course_id. After finding the course_id we need to find the creator's id to make computazion easy when the course creator addresses discounts.

We know the @course_id, thus a simple search does the trick:

SQL statement for find the course creator's id:

```
SELECT    C.course_creator_id
FROM      course C
WHERE     C.course_id = @course_id
```

After that SQL statement, course_creator_id will be stored in @course_creator_id
With that, we will have all the necessary information for inserting discounts.

SQL statement for inserting discounts in the discount entity:

```
INSERT INTO     discount(start_date, end_date, percentage,
                discounted_course_id, allower_course_creator_id)
VALUES          (CURRENT_DATE, @end_date, @percentage, @course_id,
                @course_creator_id)
```

When a discount offer will be made from the admin, course creator will be able to see them.

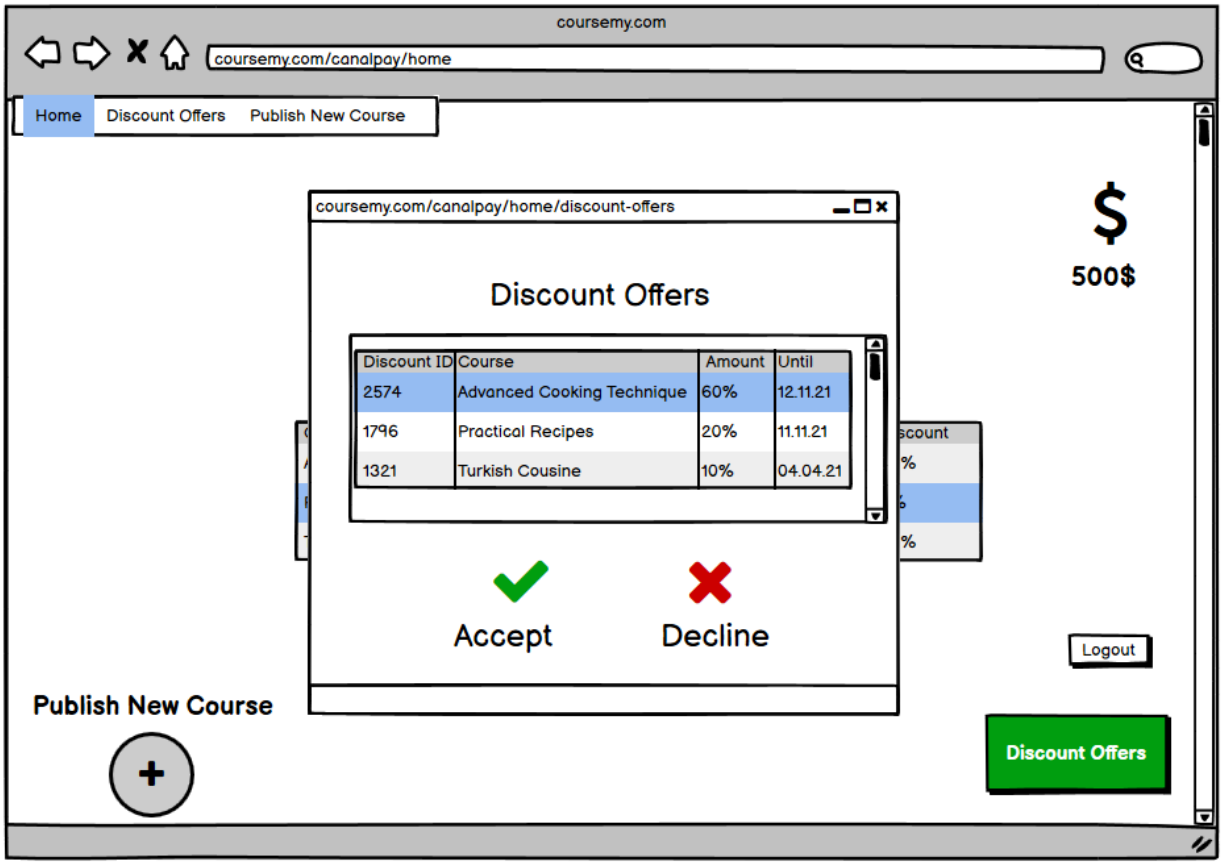

Fig. 40: Screen for seeing discount offers, from course creators' view

SQL statement for course creator to display newly offered discounts:

```
SELECT      D.discount_id, C.course_id, C.course_name, C.course_description,
            C.course_price, D.percentage
FROM        discount D, course C
WHERE       D.is_allowed = 0
```

When a discount offer is selected by the course creator we will get the discount id from the php as @discount_id. In the opened page we will display the course_name, course_description, discount percentage and course price which are already stored in the php. In the opened window we will ask the course creator's approval. When the course creator makes a decision, either one of the following SQL statements will be executed:

SQL statement when course creator approves the discount offer:

```
UPDATE      discount
SET         is_allowed = TRUE
WHERE       discount_id = @discount_id
```

SQL statement when course creator rejects the discount offer:

```
DELETE FROM     discount
WHERE           discount_id = @discount_id
```

Cancel an already applied discount

First admin can see the list of offers and discounts of the courses. To show the list of the discounts, we can simply retrieve the offers from the discount entity. Discount entity will automatically delete the offers which are expired (end_date < current_date). Thus, the discount entity only stores the current offers and discounts.

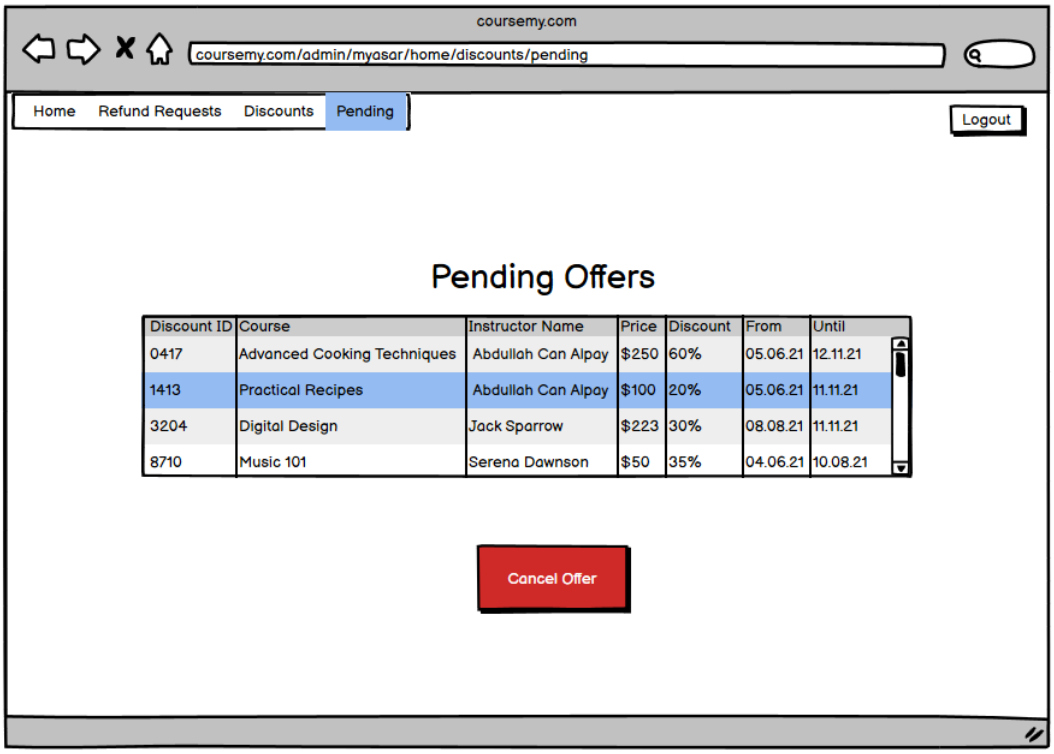

Fig. 41: Pending discount offers list

SQL statement for displaying all the offers and the discounts:

```
SELECT    D.discont_id, D.discounted_course_id, D.start_date, D.end_date,
          D.percentage, C.course_name
FROM      discount D, course C
WHERE     C.course_id = D.discounted_course_id
```

When admin selects a discount from the discount list to delete, discount will be deleted from the discount entity. We will get the discount id from the php as @discount_id.

SQL statement for deleting the selected discount or discount offer for the course:

```
DELETE FROM    discount
WHERE          discount_id = @discount_id
```

Website

https://oguzkaanimamoglu.github.io/Online-Course-Platform/