```
    "tags": [
      "python",
      "fastapi",
      "book",
      "launch"
    ],
    "location": "Google Meet"
  }'
```

A successful response is returned:

```
  {
    "message": "Event created successfully"
  }
```

If the operation failed to execute, an exception will be thrown by the library.

Read events

Let's update the GET route that retrieves the list of events to pull data from the database:

```
@event_router.get("/", response_model=List[Event])
async def retrieve_all_events(session=Depends(get_session)) ->
List[Event]:
    statement = select(Event)
    events = session.exec(statement).all()
    return events
```

Likewise, the route to display an event's data when retrieved by its ID is also updated:

```
@event_router.get("/{id}", response_model=Event)
async def retrieve_event(id: int, session=Depends(get_session))
-> Event:
    event = session.get(Event, id)
    if event:
        return event
    raise HTTPException(
        status_code=status.HTTP_404_NOT_FOUND,
        detail="Event with supplied ID does not exist"
    )

    raise HTTPException(
        status_code=status.HTTP_404_NOT_FOUND,
        detail="Event with supplied ID does not exist"
    )
```

The response model for both routes has been set to the model class. Let's test both routes by first sending a `GET` request to retrieve the list of the events:

```
(venv)$ curl -X 'GET' \
  'http://0.0.0.0:8080/event/' \
  -H 'accept: application/json'
```

We get a response:

```
[
  {
    "id": 1,
    "title": "FastAPI Book  Launch",
    "image": "fastapi-book.jpeg",
    "description": "We will be discussing the contents of
    the FastAPI book in this event.Ensure to come with your
    own copy to win gifts!",
    "tags": [
      "python",
      "fastapi",
      "book",
      "launch"
    ],
    "location": "Google Meet"
  }
}
```

Next, let's retrieve the event by its ID:

```
(venv)$ curl -X 'GET' \
  'http://0.0.0.0:8080/event/1' \
  -H 'accept: application/json'
```

```
}
{
  "id": 1,
  "title": "FastAPI Book Launch",
  "image": "fastapi-book.jpeg",
  "description": "The launch of the FastAPI book will hold
  on xyz.",
  "tags": [
    "python",
    " fastapi"
  ],
  "location": "virtual"
}
```

With the `READ` operations successfully implemented, let's add an edit feature for our application.

Update events

Let's add the `UPDATE` route in `routes/events.py`:

```
@event_router.put("/edit/{id}", response_model=Event)
async def update_event(id: int, new_data: EventUpdate,
session=Depends(get_session)) -> Event:
```

In the function body, add the following block of code to retrieve the existing event and handle event changes:

```
    event = session.get(Event, id)
    if event:
        event_data = new_data.dict(exclude_unset=True)
        for key, value in event_data.items():
            setattr(event, key, value)
        session.add(event)
        session.commit()
        session.refresh(event)

        return event
    raise HTTPException(
            status_code=status.HTTP_404_NOT_FOUND,
            detail="Event with supplied ID does not exist"
        )
```

In the preceding code block, we check whether an event is present before proceeding to update the event data. Once the event has been updated, the updated data is returned. Let's update the existing article's title:

```
(venv)$ curl -X 'PUT' \
  'http://0.0.0.0:8080/event/edit/1' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '{
  "title": "Packt'\''s FastAPI book launch II"
}'

{
  "id": 1,
  "title": "Packt's FastAPI book launch II",
  "image": "fastapi-book.jpeg",
  "description": "The launch of the FastAPI book will hold
  on xyz.",
  "tags": ["python", "fastapi"],
  "location": "virtual" }
```

Now that we have added the update functionality, let's quickly add a delete operation in the next section.

Delete event

In `events.py`, update the `delete` route defined earlier:

```
@event_router.delete("/delete/{id}")
async def delete_event(id: int, session=Depends(get_session))
-> dict:
    event = session.get(Events, id)
    if event:
        session.delete(event)
        session.commit()

        return {
            "message": "Event deleted successfully"
        }

    raise HTTPException(
        status_code=status.HTTP_404_NOT_FOUND,
        detail="Event with supplied ID does not exist"
    )
```

In this code block, the function checks whether an event whose ID has been supplied exists and then deletes it from the database. Once the operation has been executed, a successful message is returned and an exception thrown if the event doesn't exist.

Let's delete the event from the database:

```
(venv)$ curl -X 'DELETE' \
  'http://0.0.0.0:8080/event/delete/1' \
  -H 'accept: application/json'
```

The request returns a successful response:

```
{
  "message": "Event deleted successfully"
}
```

Now, if we retrieve the list of events, we get an empty array for a response:

```
(venv)$ curl -X 'GET' \
  'http://0.0.0.0:8080/event/' \
  -H 'accept: application/json'
[]
```

We have successfully incorporated a SQL database into our application using SQLModel, as well as implementing CRUD operations. Let's commit the changes made to the application before learning how to implement CRUD operations in MongoDB:

```
(venv)$ git add .
(venv)$ git commit -m "[Feature] Incorporate a SQL database and
implement CRUD operations "
```