Activities 7: Application design

Table of contents

Activities 7: Application design

Theme: Designing applications

Instructions (7-0-instructions.md) are in the docs/7_app_design folder:

1. Introduction to application design (7-01-introduction.md)
2. Examples of application designs (7-02-diagram-examples.md)
3. Identify classes (7-03-identify-classes.md)
4. Draw the application design diagram (7-04-draw-design.md)
5. Review the design (7-05-review-design.md)
6. Draw the application design for the paralympics prediction web app (7-06-design-medals.md)
7. Results of using genAI for the application design (7-07-genAI.md)

Next activity (7-01-introduction.md)

1. Introduction to application design

Design principles

This is covered in the lecture notes with references for more information in the reading list.

Summary of some common design principles:

- KISS: prefer simple to complex. Follow standards that make code easier to read. Deconstruct problems into smaller ones; give each function one thing to do. Use the least complex algorithm, data structure, etc.
- DRY: do not repeat business logic; some code may be repeated if it relates to different business logic.
- Loose coupling: Coupling is the degree of interdependence between software modules. Keep dependencies between modules/classes to the minimum needed.
- High cohesion: Cohesion refers to how strongly related and focused are the various responsibilities of a module; and how they work together to create something more valuable than the individual parts. Suggestions to increase cohesion:
    - The functionalities embedded in a class, accessed through its methods, have much in common.
    - Methods carry out a small number of related activities.
- Separation of concerns: A design principle for separating an application into distinct sections such that each section addresses a separate concern. A concern is a set of information that affects the code of a computer program.
- Modular: separate code into modules. The functions within a module should be cohesive. Modules should be loosely coupled.

There are many more sets of principles such as SOLID, GRASP if you want to try and apply these!

Many of the principles relate to object-oriented design. While your design will have some classes, you may also have functions that do not belong to a class. In Python, functions can be grouped in modules.

Consider the principles when you design your app.

Design patterns

One way to follow these principles in your application's design is to apply relevant design patterns.

A design pattern is a general blueprint to solve a common problem. First documented as design patterns in the book Design Patterns: Elements of Reusable Object-Oriented Software in 1994 (https://en.wikipedia.org/wiki/Design_Patterns).

There are now many patterns, and you can find Python examples e.g. see those in Refactoring Guru (https://refactoring.guru/design-patterns/python).

Some of the patterns that may be relevant to your coursework:

- Model view controller (MVC): application design pattern for apps with a user interface that separates business logic and views from the underlying data. This Real Python tutorial (https://realpython.com/lego-model-view-controller-python/) explains MVC using Python.
- REST (or RESTful) API: REpresentational State Transfer (REST) was first discussed by Roy Fielding. This is a software architecture style for client and server communications using HTTP. REST APIs are often used to access data as web service. This Real Python tutorial (https://realpython.com/api-integration-in-python/) gives an explanation of Python REST APIs.
- Object relational mapper (ORM): An ORM (https://www.fullstackpython.com/object-relational-mappers-orms.html) provides an abstraction that maps Python classes to relational database tables. SQLModel follows the ORM pattern.

Optionally, you may consider design patterns, for all or part of your application.

Documenting the application design

To design the app, you are likely to consider the requirements, wireframes and data. Determine what your code needs to be able to do. Then consider how to structure that code in line consistent with design principles. Think about the Python packages, modules, functions and classes that will be needed.

Having done that, you then need to document that as a diagram. Yet unlike the database design, there is no prescribed diagram format. The next activity offers some examples for you to reflect on.

Specific techniques that you can read about include:

1. UML class diagrams (https://realpython.com/lessons/uml-diagrams/), or variants of these.

   Most examples of class diagrams assume an object-oriented design. You are not expected to apply a fully object-oriented design.

   UML also has its critics, with some seeing it as overly complex.

   The assessment considers the clarity of the design and the application architecture. Diagrams should be easy to read, though strict adherence to UML is not assessed.

   The class diagram focuses on classes, so you may need to adapt it to include packages, modules and functions.

2. C4 model (https://c4model.com/) is a more recent approach to visualizing software architecture.

   C4 describes a set of hierarchical diagrams, rather than notation.

   - Level 1: Context - System context diagram
   - Level 2: Containers - Container diagram
   - Level 3: Components - Component diagram
   - Level 4: Code diagram

   For the coursework you will need the level of detail shown at level 4.

Tools for diagrams

For the coursework all diagrams can be hand drawn and do not need to be created with a diagramming app.

Tablet drawing apps, or apps such as PowerPoint can be used.

Mermaid is used in the course materials. Most digrams can be created in the Python version, though you may need to go Mermaid online (https://mermaid.live/edit#pako:eNqdVV1vmzAU_StXfuoklpHQfKFpUghptYdK3dq9TEiVASexCnZkmzVZlf8-YyCQxGm38ZLYnONzj--xeUUJTwny0fx6zpkiWxUx0I-iKiPwsJOK5FC_gZTilcA5LLmAr3pKMKIgwOyZslWNrdiL8uVGUEmeAl6wFIvdVew6EKES3Uy5EfoArxWjfO6JkJxdJYvG-0Fa06c1X9zMy2XupeJZxl_EUe1CUoDr8oeQHKdOdzrChngGNeqLIqKo7VHcAshRw_E9jgXU4sBVnj0T-Nx0k6WidhbHpQDRYl7Q7r0nQyidXPg-KCSMBZ1mxkoidM1SX03NXBudNxpQRmEiclTI-ISk5dtQV2XA1OXQ0sts6b1M1FfQhnR1m00gILLWjCJKvxX58WYOWRzygjvTPl_QXfbVvMwVh8zDHNauWqjketTU34SiS-nti38W0EKUtd-Y6n9BgzicuG9N0JgqJ0W3FoEbt8T-IcbtA7Cyd9DNAL6nWTdy7x7__yQRB4Wq5Dt687BPsLZUCZnD4Slso6OrM_83eO9hajhR5-XI6a-5zQFOWglaIp8JQriII3UL_UQma5GSO9UrjfQNxf8EheZilDE9pq2wewn53nDFLxYrZG_xJnUo2KTYkXC6gt6gGh5_NqNubjIeufgYjdzx2B8P-_g9eH4F0) for the C4 diagrams.

There are online tools such as LucidChart, you may need to create an account, and free versions may limit the number of diagrams you can create.

Tools specific to C4:

- Structurizr (https://www.structurizr.com/)
- Structuizer for VSCode (https://marketplace.visualstudio.com/items?itemName=systemticks.c4-dsl-extension)

For apps that have already been written, you can generate a UML diagram from the code. See PyCharm UML class diagrams (https://www.jetbrains.com/help/pycharm/class-diagram.html) or search for a VS Code extension (https://marketplace.visualstudio.com/search?term=UML&target=VSCode&category=All%20categories&sortBy=Relevance).

Next activity (7-03-identify-classes.md)

Application design diagram examples

These examples are intended to make you think about your app and how you might want to design its structure. There is no single correct way. For the coursework, it is important that you are able to explain why you chose to design your app in a given way.

Note that simply repeating my comments below in your coursework is not appropriate.

REST API (UML)

An application programming interface (API) defines the rules that you must follow to communicate with other software systems. A REST or RESTful API is an interface that two computer systems use to exchange information over the internet.

REST is an architectural style for designing networked applications to communicate over the internet using standard HTTP methods.

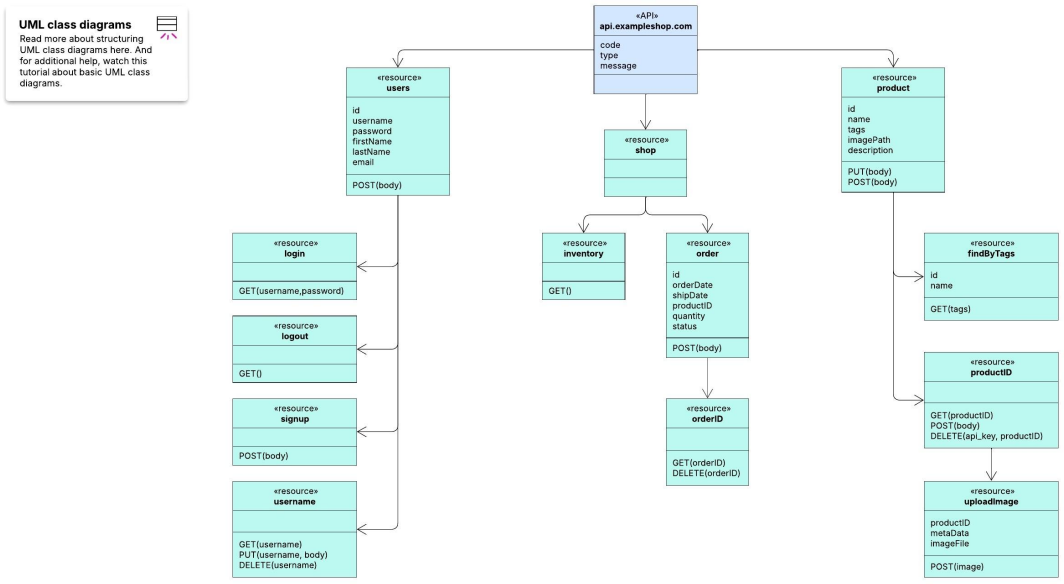
A REST API allows clients (like web or mobile apps) to interact with servers using HTTP requests.

Common HTTP methods used:

- GET – retrieve data
- POST – send data to create something
- PUT – update existing data
- DELETE – remove data

Each resource (like a user, product, or article) is typically represented by a URL, and responses are usually in JSON format.

Consider this class diagram for a REST api.



0-1. Class diagram of a REST API

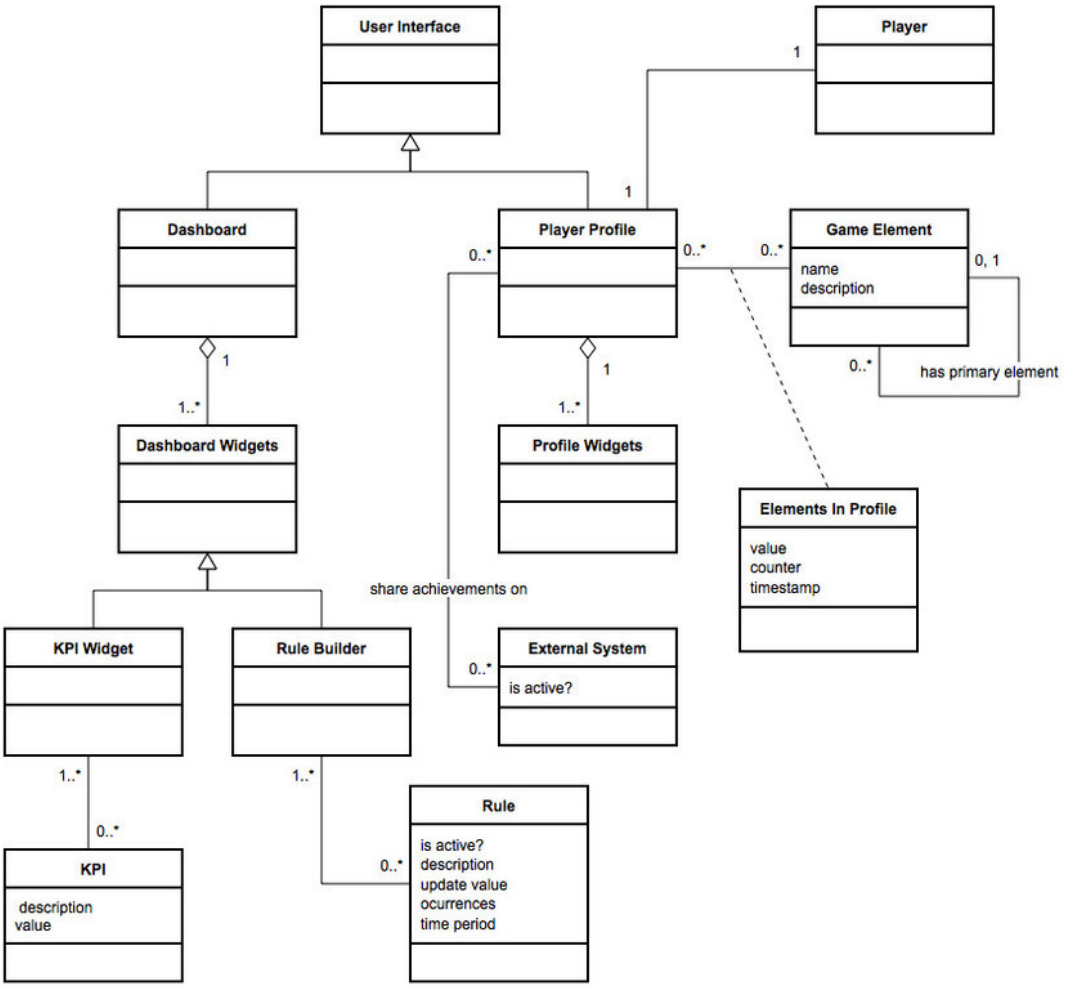
Source: Lucid (https://lucid.co/templates/uml-api-diagram-example)

The diagram shows how the REST HTTP methods, with route (url) names and the data attributes. This design is useful to another developer who wants to make use of information about a product, or an order, etc. in their own application.

It does not, however, describe an online shopping app. The methods for a user to search a product, place an order etc. are not represented. You would need an additional diagram.

Dashboard app (UML)

This next diagram from Jorge Simoes (https://www.researchgate.net/figure/Dashboard-and-Player-Profile-Class-Diagram_fig31_292720849) shows the design of a dashboard application. The classes in the diagram don't show the attributes and methods (which you would need to include for the coursework). However, it shows how he intends to structure the app code.



0-1. Class diagram of dashboard and player

C4

C4 describes software architecture at 4 levels, and the diagrams are used in combination.

To view, have a look at some of the examples online:

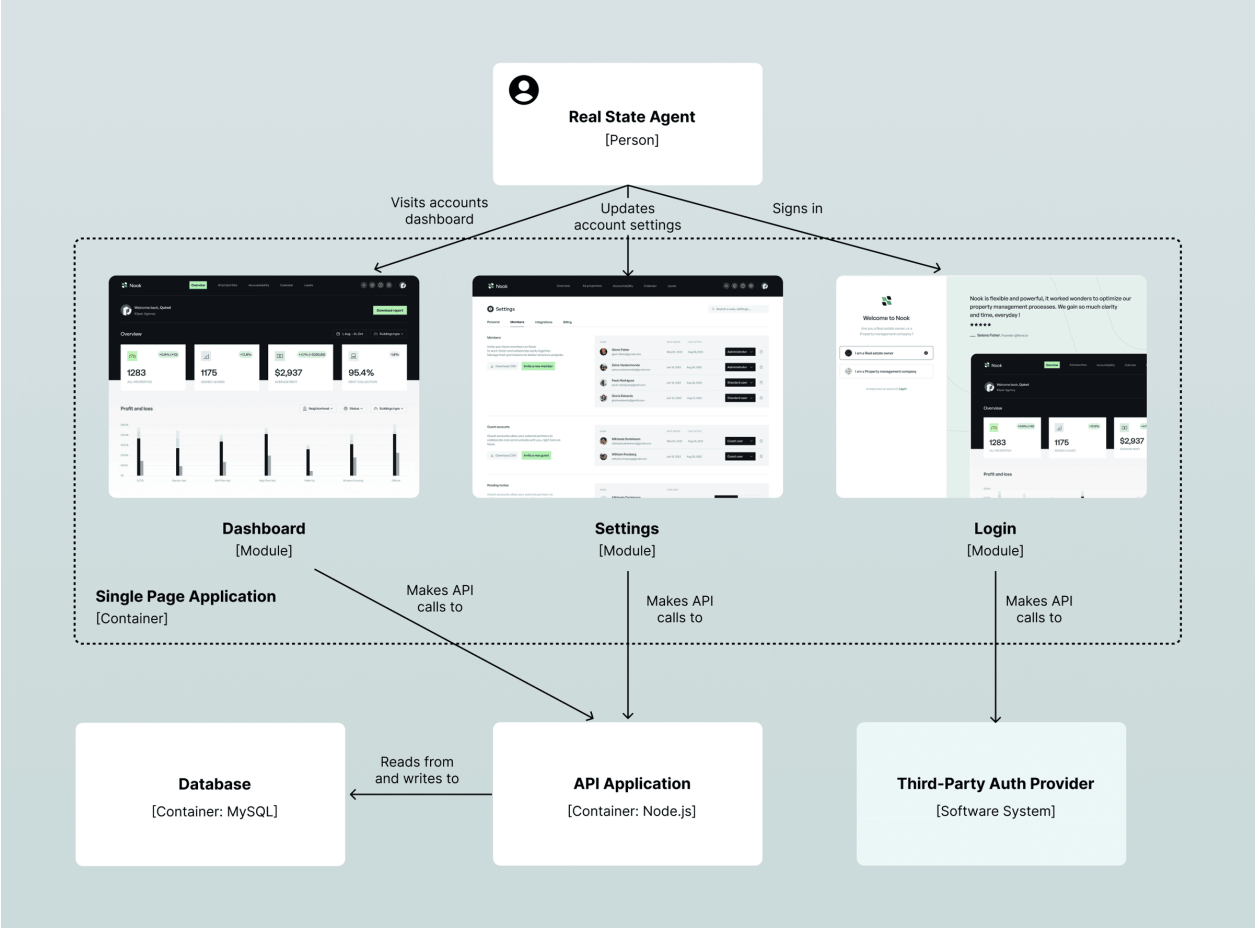
- Freecodecamp tutorial (https://www.freecodecamp.org/news/how-to-create-software-architecture-diagrams-using-the-c4-model/#heading-diagrams-as-code) and GitHub repo displaying the diagrams (https://plutov.github.io/c4-diagram-example/master/task-management-system/context/)
- Dev.to example of C4 diagrams for Google Maps (https://dev.to/anwaar/c4-model-real-world-example-with-google-maps-3ano)

There are few examples that illustrate the code level; for the coursework you need to include code level diagram (but not code).

Other (neither C4 nor UML)

‘Module’

The author describes this as a module diagram for a dashboard web app.



Source: Frontend at scale (https://frontendatscale.com/issues/17/)

The diagram shows how the dashboard generally relates to the API, though it omits detail such as what the functions are that are needed for the features of the dashboard.

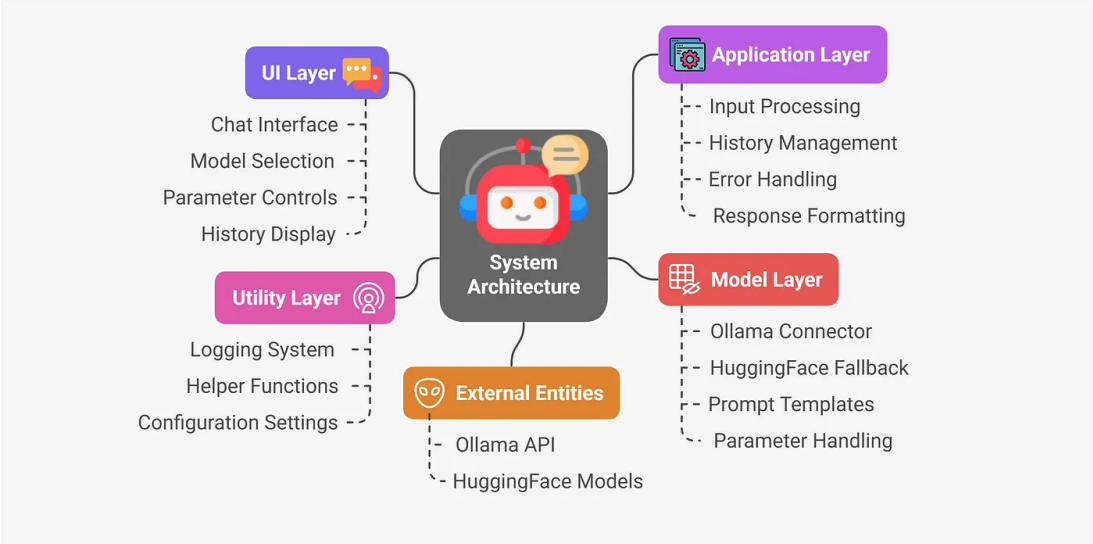
It uses the wireframes in the diagram but doesn’t have to. The author notes these could be rectangles.

This gives a clear picture of the overall structure but lacks enough details for you to determine if the design of the code meets principles of good design. You could consider combining this with a further diagram to show the next level of detail.

Layer

the following is from Shanoj on Medium (https://blog.stackademic.com/build-a-local-llm-powered-q-a-assistant-with-python-ollama-streamlit-no-cloud-required-800af31ce0bb)

This diagram represents a design separated into layers.



0-1. Architecture in layers

Again, it lacks the detail of what is in each component, e.g. what functions or classes does the chat interface? What are the parameters included in the parameter controls? Nevertheless, the overall design pattern is clear from the diagram.

Streamlit multi page dashboard app

This blog post (https://blog.streamlit.io/how-to-build-an-interconnected-multi-page-streamlit-app/#code-structure-structuring-for-scalability-and-modularity) is quite long and does not have an overall diagram, yet shows how other techniques can be used.

The table in the 'session states' section shows how a table could be used rather than a more visual diagram to show the attributes and functions needed for each page in a multipage dashboard.

The models in this article are shown as terminal style text rather than a class diagram. It may be a personal opinion, but I found this format harder to read than a class-style rectangle with attributes.

Summary

Whichever technique you use to represent the design, the most important aspect of the application design is the thought and reasoning in determining the structure of the app and the contents. The next activities walk through activities that can help you decide what the app needs to include.

Next activity (7-03-identify-classes.md)

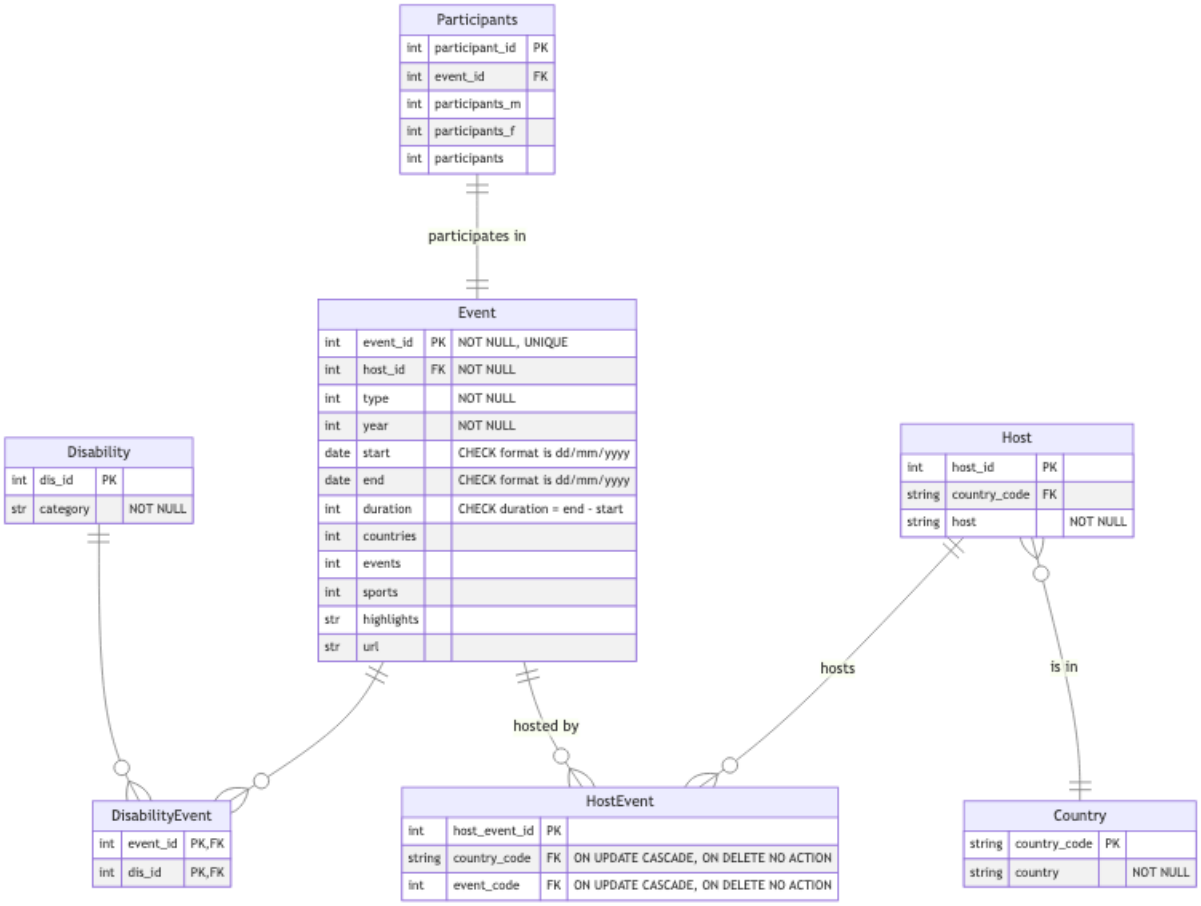
2. Application design: Identify Python classes and functions

Identify the Python classes

Identify classes from the database design

You already documented a database design using the ERD format. This is a good starting point for identifying classes.

The version of the ERD below uses all the raw data, and not the reduced set from week 3.



This ERD was written before we had gathered the requirements. The app will need more classes than just those that map to the stored data.

Considering the ORM pattern, each of the tables also represents a potential Python class:

- Event: type, year, start, end, duration, countries, events, sports, highlights, url
- Country: country_code, country_name
- Disability: category
- Host: host_name
- Participants: participants_m, participants_f, participants

From the requirements

Another way to identify the potential classes is to read the requirements (user stories in this case) and apply a data driven design approach:

- Nouns and noun phrases in the requirements suggest classes
- Adjectives suggest an attribute of a class
- Verbs and verb phrases suggest functions

In this activity the application design considers all the requirements. For the coursework you may limit your design to cover only the 'Must' and 'Should' have requirements.

Read the user stories below and underline:

- Nouns and noun phrases in the requirements suggest classes
- Adjectives suggest an attribute of a class
- Verbs and verb phrases suggest functions

Ref	*User story*	*Constraints*	*Priority*
3.	As a student I want to browse information about paralympic games so that I can see what is available.		Must
5.	As a student I want to view statistical information about the paralympics for each event and across events so that I can find information for my project.	Must include number of sports, participants and events.	Must
12.	As a student or teacher I want to be able to access the quiz app from a PC or mobile phone so that I can work at home or in school.		Must
13.	As a student or teacher I want to be able to use the app through a web browser so that I do not have to install an application on my device.	Supports the 3 most frequently used browsers in the UK region.	Must
2.	As a system administrator, I want to make sure that only a teacher can edit questions and quizzes, and access student scores.	Authentication must use the school's single sign on method.	Should
4.	As a student I want to search for information so that I can answer quiz questions.	Search should find data in any of the tables related to the events.	Should
6.	As a student I want to see charts that show trends over time in the events data and be able to refine the charts to specific fields or date ranges so that I can find information for my project.	Support filter by date range, event type (summer/winter), gender and disabilities included. Chart types include line, bar and map.	Should
7.	As a student I want to complete a quiz so that I can assess my learning.		Should

Ref	*User story*	*Constraints*	*Priority*
9.	As a teacher I want to add and edit questions so that these can be added to quizzes.	Allow multiple choice question type.	Should
10.	As a teacher I want to add and edit quizzes, including adding questions to them, so that I can offer my students formative assessment.	A question can be added to more than one quiz. A question has multiple answers. An answer is correct or incorrect. A question has a score.	Should
8.	As a student I want to submit my quiz score so that I can participate in the competition to see who scores the highest.		Could
11.	As a teacher I want to access students' scores to see which student gained the highest score.	Display student first name, last name and score. Optionally allow download to .txt file.	Could
1.	As a system administrator I want to make updates to the paralympic games database so that the data remains accurate and up to date.	Updates will be requested once per day. A request will be triggered in the quiz app and will use the paralympics API to look for changes since the last date of update. Any changes will be updated in the database. The last update date will be stored.	Won't

A partial list of nouns and adjectives:

- Student
- Teacher
- Statistic: sports, participants, events
- Chart: chart type
- Quiz: name

A partial list of verbs and verb phrases:

- browse information
- view statistics
- answer question
- search
- see chart
- complete quiz

Read the user stories and complete the lists above.

Identifying classes from use cases

This is not covered in the teaching materials. However, if you create use cases in the coursework this guide to identifying an MVC design from use cases (https://guides.visual-paradigm.com/from-use-case-to-mvc-framework-a-guide-object-oriented-system-development/) may be useful.

Review the list of classes, attributes and functions

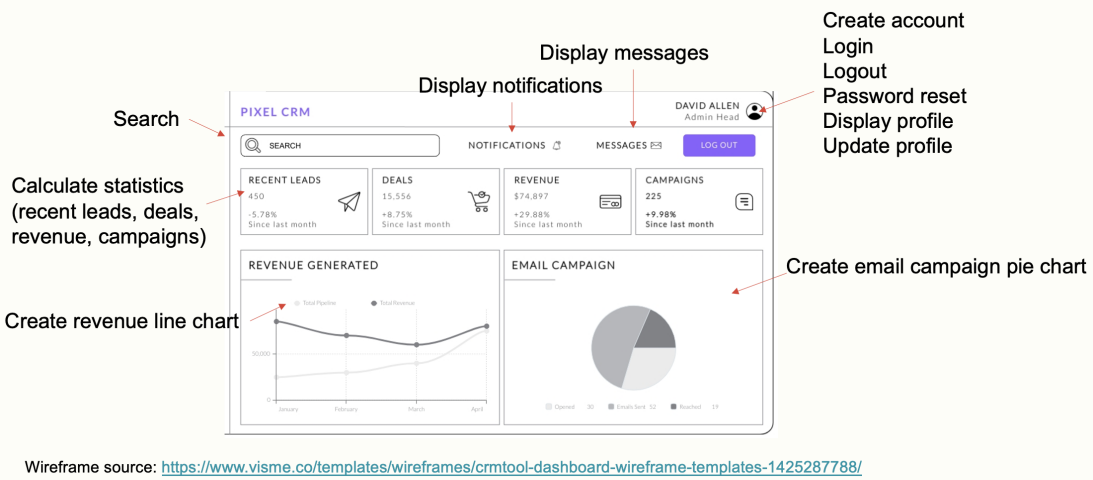
> Review the list of classes and their attributes. You will need to complete the following with the results of the previous step:

- Event: type, year, start, end, duration, countries, events, sports, highlights, url
- Country: country_code, country_name
- Disability: category
- Host: host_name
- Participants: participants_m, participants_f, participants
- Student
- Teacher
- Statistic: sports, participants, events
- Chart: chart type
- Quiz: name

Remove duplicates (e.g., Statistic and its attributes is already covered by the Event class in the ERD, though you could consider splitting the event into two classes).

Consider any missing attributes and add them.

You could also review the classes and functions against the wireframes to check for completeness, e.g.



0-1. Identifying potential functions from a wireframe

Review the list of functions. You will need to complete the following with the results of the previous step:

- view_webpage (replaced 'browse_information')
- view_statistics
- answer_question
- search
- view_chart
- complete_quiz

Next activity (7-04-draw-design.md)

3. Determine the application structure

Model-View-Controller (MVC) MVC is a common design pattern for web applications. This activity assumes a decision has been made to structure the app in accordance with the MVC pattern. You are not required to follow this pattern in your coursework. It is used here for speed in the tutorial, so you don't have to start from a blank sheet in determining the app structure.

MVC design pattern

The goal of the MVC design pattern is to separate concerns in a web application:

- the **model** handles data and business logic,
- the **view** manages the user interface and presentation,
- and the **controller** acts as an intermediary between them.

Key concepts:

Routes and controllers

Each *route* in a web app has a *controller* action. When a user entera a URL, the application attempts to match it to a defined route, and, if successful, calls that route's controller action. The controller typically: - Retrieves data from the model (e.g., via a database) - Passes that data to a view, which renders the page

Models

Represent the data and its associated logic. They interact with the database and determine what data is sent to the controller. You have already seen the ORM data mapper pattern to map between these model classes and the database tables.

Views

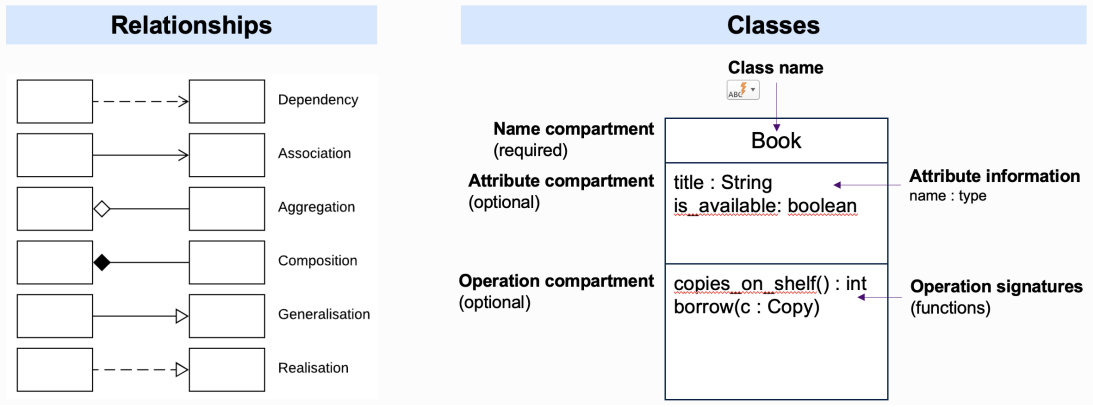
In the view, the data is accessed and the information contained within is used to render the HTML content of the page the user ultimately sees in their browser. You already designed the views as wireframes.

Diagram format

This activity is loosely based on the UML class diagram format. This is not the only choice of diagram.

Mermaid (https://mermaid.js.org/ecosystem/tutorials.html) is used as you likely installed to view the database diagrams. If not, you may need to view these activities on GitHub.

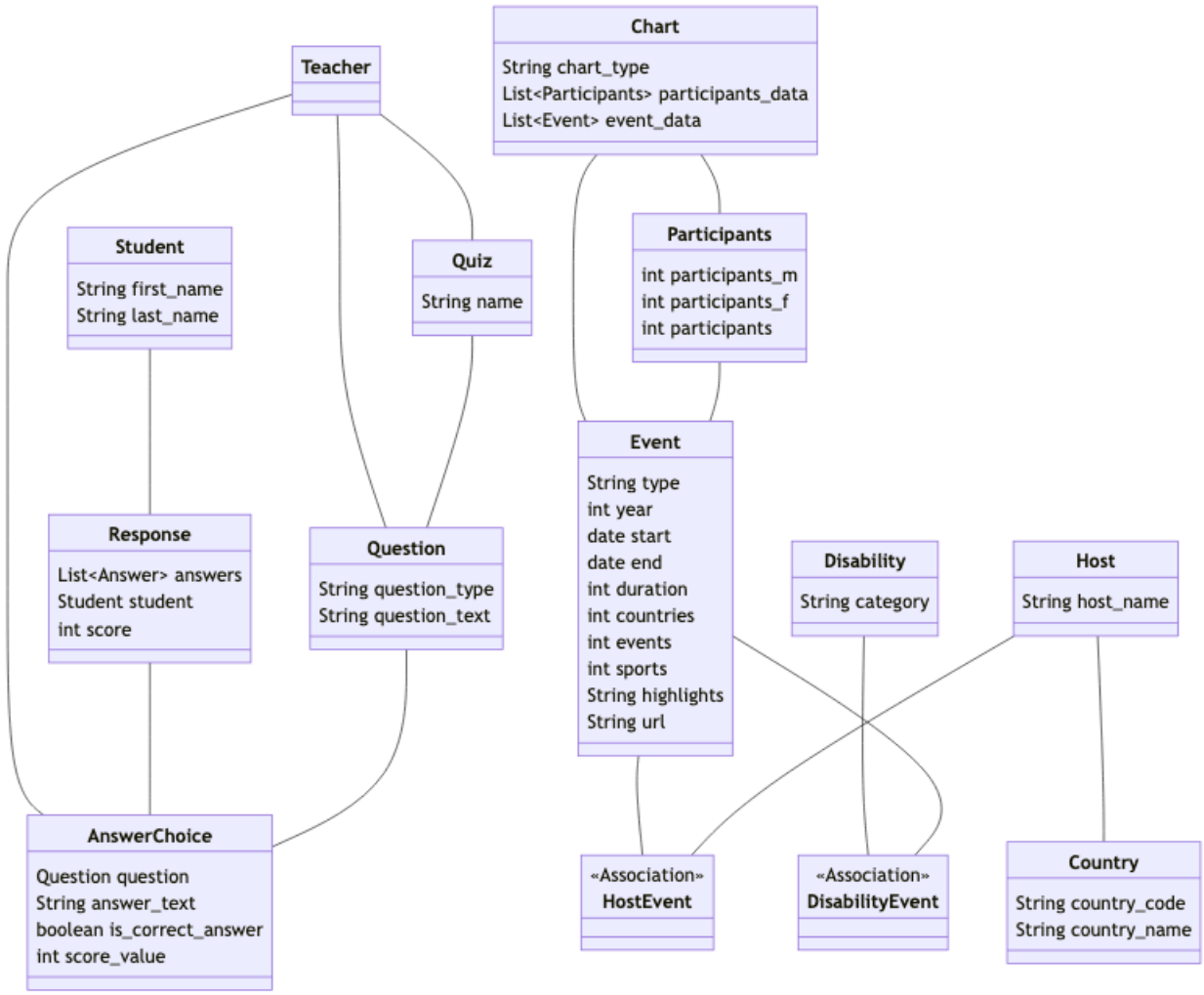
Remember the following symbols for classes and relationships:



0-1. UML class diagram notation

Add the model classes

The following were derived from the ERD and the activities to identify classes from the user stories.

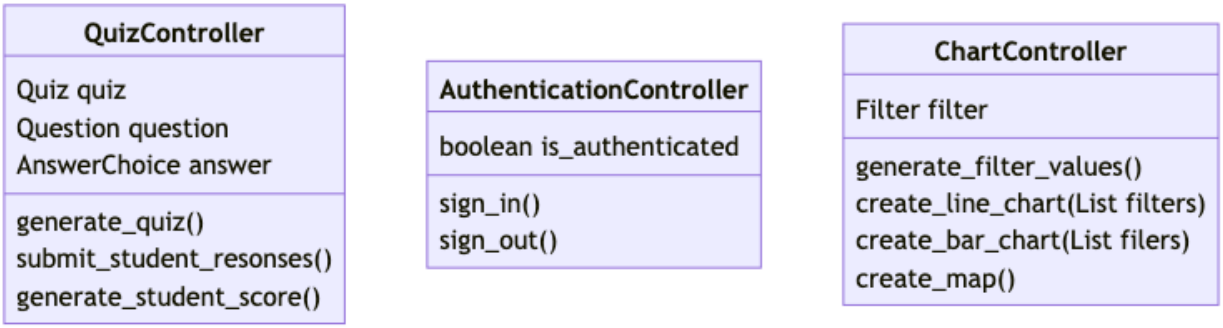


Add the controller classes

These control the business logic. I have grouped them according to function; you may choose a different way to group them. There is no single way to design these!

If not following the MVC model, these operations may be in the relevant model classes.

The logic to create, read, update and delete any of the model classes will be handled by the ORM, so has not been added here. You can add them to the class diagram if you prefer.

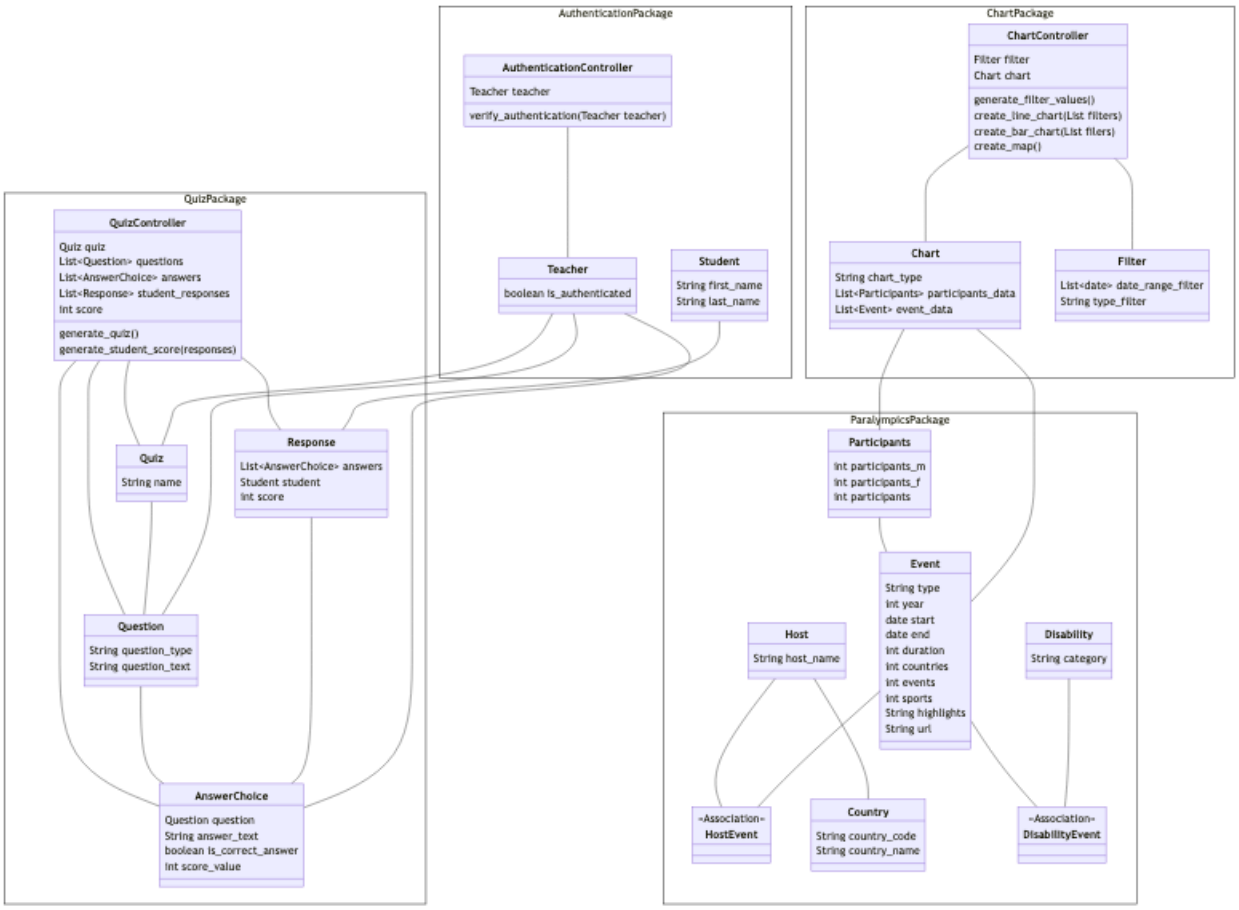


Add all the classes to the class diagram and add the relationships

The 'view' classes have been omitted in the diagram below. In the coursework you have represented these with the wireframes.

You may not agree with this diagram, you may have made different choices to structure it differently.

This version also groups the Python classes into Python packages. This is not required.



Next activity (7-05-review-design.md)

4. Review the design

This is not easy as you are trying to critically evaluate our own work against a general design principle or pattern.

As a starting point, look at the diagram you have created and consider:

- Are the relationships between classes, modules, functions only those that are needed?
- Are the attributes and methods/functions within a class or module cohesive? If not, do you need to split them into more classes, functions, modules.
- Does the structure seem clear (simple as possible)?
- Would the structure allow you to reuse components in another application?
- Is the same business logic represented in more than one place? If so, then remove the duplication. You don't have code at this point, however, you may have similar functions (methods, operations) that indicate duplicated logic.

This is not an exhaustive list of questions. This aspect is challenging.

Design principles to look up and consider:

- Separation of Concerns (SoC)
- Modularity
- Single Responsibility Principle (SRP)
- DRY (Don't Repeat Yourself)
- KISS (Keep It Simple, Stupid)
- YAGNI (You Aren't 'Gonna' Need It)
- Encapsulation
- Loose Coupling & High Cohesion

Next activity (7-06-design-medals.md)

6. Draw the application design for the paralympics prediction web app

This is an optional activity to give you the opportunity to apply the application design techniques with less instruction.

Project overview

	Paralympics prediction app
Target audience	UK school pupils aged 16 to 18. These students are used to using mobile phones and web based apps.
App overview	A web app that includes a machine learning prediction algorithm to predict country medal results in future paralympic games. Historical medal rankings will be included in the app. The app will use the paralympics dataset medal rankings; and a machine learning model trained on the dataset.
Scenario	Students have been asked to learn about the medal rankings in previous paralympic games, and to make predictions for the next paralympic games including:

User stories

This is an abbreviated version of user stories for the design activity. The "... so that ...", prioritisation and reference number have been omitted.

- As a student I want to see the medal results table for each paralympics. I want to be able to filter the list for the top 10.
- As a student, I want to enter the country name and future year and get a prediction of the gold, silver and bronze medals for that country in the specified paralympics.
- As a student, I want to be able to predict the top 10 countries in the next paralympics.
- As a system administrator, I want to add new data to the database when future paralympics take place.

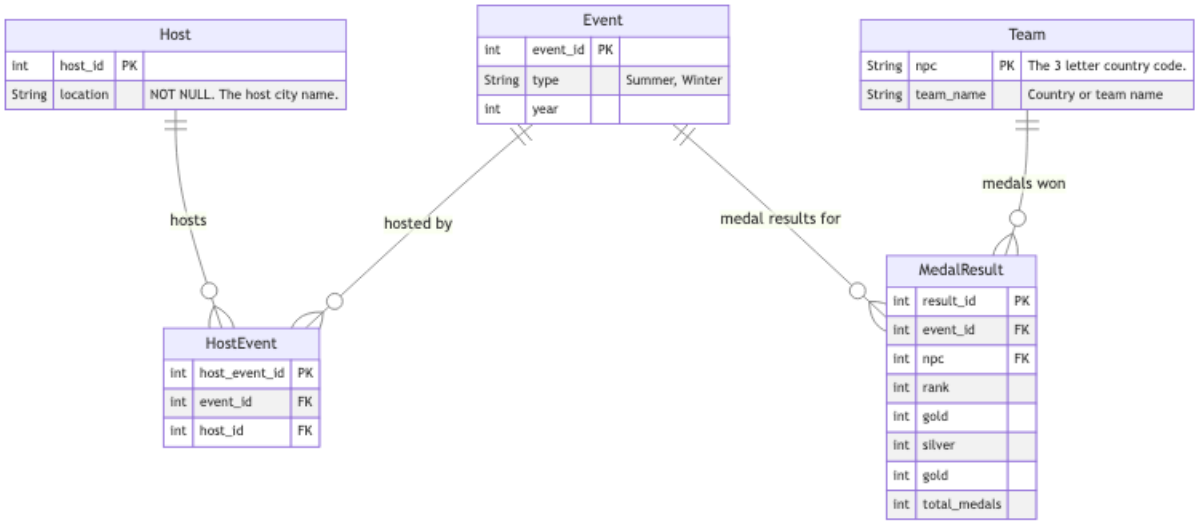
For the activity, assume that it is possible to create machine learning models that can predict the results.

ERD

The medal_standings worksheet in paralympics_all_raw.xlsx (../../src/activities/data/paralympics_all_raw.xlsx) has the following columns:

- Location: name of the paralympics host
- Year: year
- Rank: rank of the team
- Team: country, or team, name
- NPC: country code
- Gold: number of gold medals
- Silver: number of silver medals
- Bronze: number of bronze medals
- Total: total number of medals (bronze, silver and gold)

The ERD might look like this:



Create an application design

Refer to activities 1 to 4 to help you.

1. Review the user stories to identify potential classes and their attributes and operations (functions). Underline nouns (potential classes or attributes), verbs or verb phrases (potential functions).
2. Review the classes in the ERD.
3. Combine the classes and review for duplicates.
4. Draw the class diagram. Add the classes. Draw the relationships between the classes.
5. Review the diagram against the design principles. Make changes if required.

Next activity (7-07-genAI.md)

7. Generative AI in application design

Note: this activity was created 25/09/25. Repeating the activity will likely give a different result.

Tools used

The following tools were tried in relation to this week's activities:

- Goblin Tools professor (https://goblin.tools/Professor) - for summary explanations of design concepts. Prompts used:
    - `Explain the DRY design principle`
    - `Explain the low coupling design principle`
    - `Explain the high cohesion design principle`
    - `Explain the purpose of a UML class diagram`
    - The responses generated at the time were concise and consistent with teaching materials. Responses were illustrated with an example.
- Microsoft copilot - accessed using UCL account (https://copilot.microsoft.com)
    - The prompt used was: `generate a class diagram from the attached user stories` with user-stories.md (7-08-user-stories.md) attached.
    - The response is below and includes classes and attributes, and relationships. It did not identify class operations (functions). I entered a subsequent prompt `add methods to all classes`
    - A following prompt was then entered: `To what extent does this class diagram meet the principles of low coupling and high cohesion?` and again the response is below.
- ChatGPT - free 'everyday tasks' version (https://chatgpt.com)
    - the same prompt was used as for copilot. The result included methods as well as classes and attributes, and relationships between classes.
    - ChatGPT offered to draw this as a diagram, but the result was not meaningful.
    - A second prompt was used: `To what extent does this class diagram meet the principles of low coupling and high cohesion?`. The response to this was detailed and is included below.
    - A third prompt was used: `Modify the design so that it is consistent with the model view controller design pattern`. Again, the response is below.

As a learning outcome, it is recommended that you first attempt the activities without using gen AI tools as you need to learn the technique before you can critically judge the output of a gen AI tool. Comparing your own work to the response of a gen AI tool may give suggested areas where you could review your design.

copilot response

Key Classes and Relationships

1. User

   Attributes: userId, firstName, lastName, role (Student, Teacher, Admin)

   Relationships:

    - A User can complete many Quizzes

- A Teacher can create/edit Questions and Quizzes
- A User has many Scores

2. Quiz

   Attributes: quizId, title, createdBy, createdAt

   Relationships:

   - A Quiz has many Questions
   - A Quiz has many Scores

3. Question Attributes: questionId, text, score Relationships:

   - A Question can belong to many Quizzes
   - A Question has many Answers

4. Answer

   Attributes: answerId, text, isCorrect

   Relationships:

   - Belongs to one Question

5. Score Attributes: scoreId, userId, quizId, value Relationships:

   - Belongs to one User and one Quiz

6. Event

   Attributes: eventId, name, type (Summer/Winter), gender, disabilityType, year

   Relationships:

   - Used for browsing, searching, and charting

7. Chart

   Attributes: chartId, type (line, bar, map), filters

   Relationships:

   - Uses data from Event

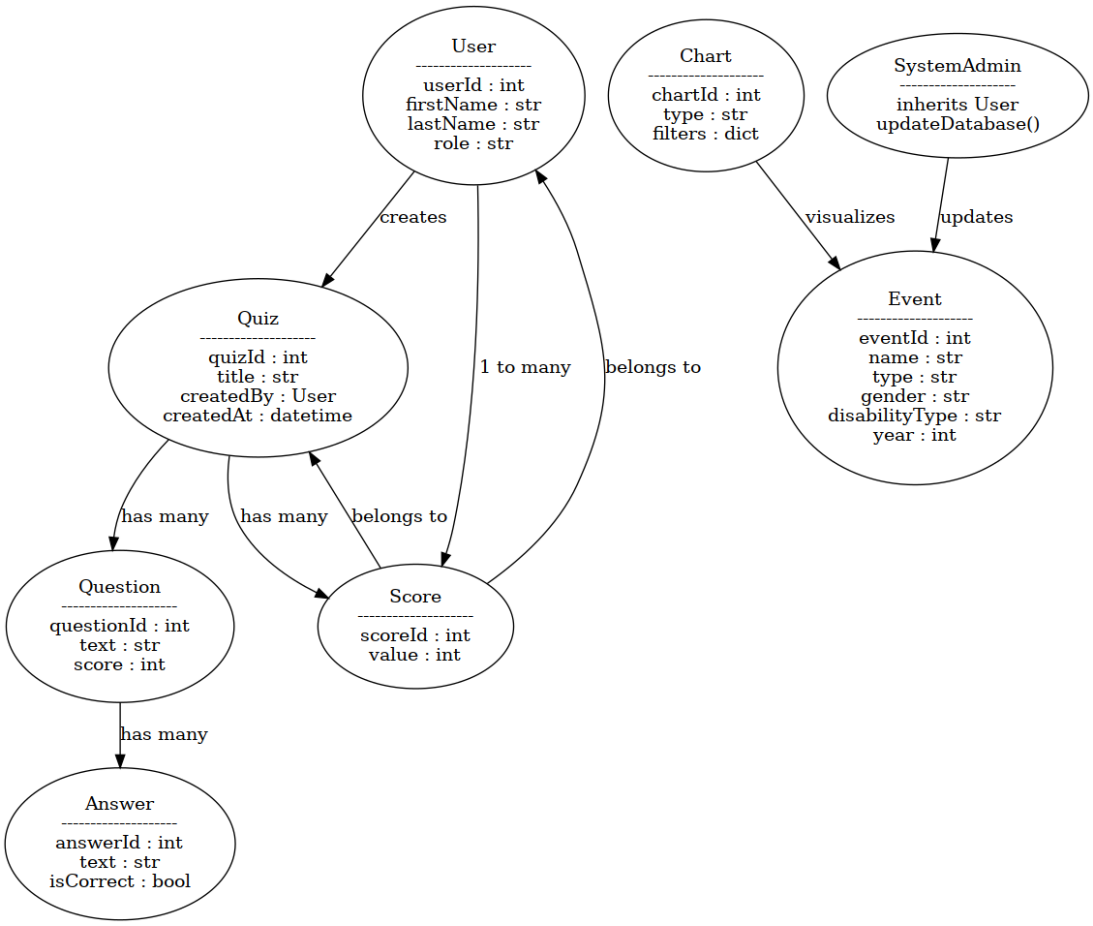
8. SystemAdmin

   Inherits from User

   Responsibilities:

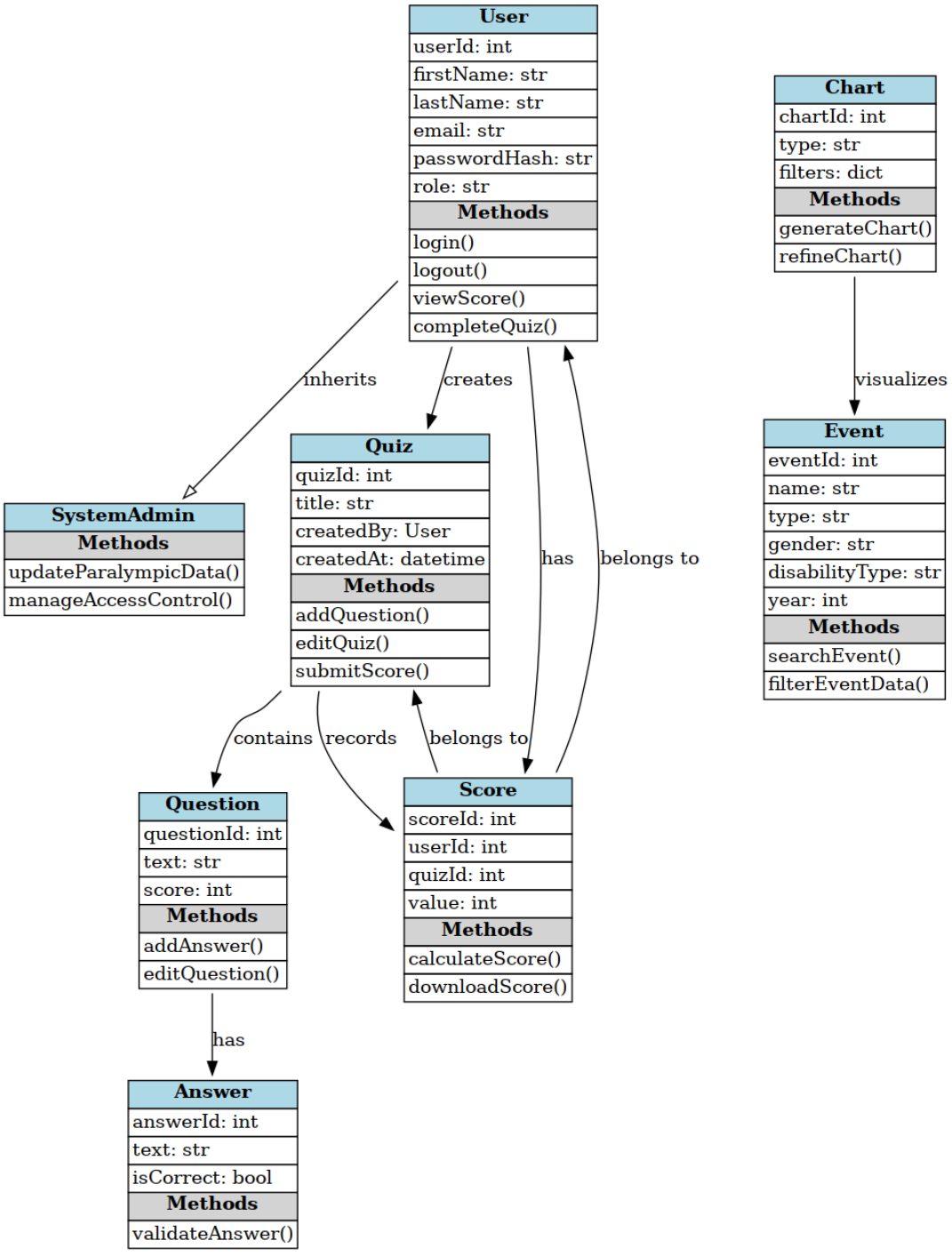
   - Update Paralympic data via API
   - Manage authentication and access control

Here is the UML class diagram generated from the user stories in your file. It models the key entities and their relationships in the quiz and Paralympic data system:



0-1. AI generated diagram

Version after methods were added:



0-1. AI generated diagram version 2

Note This is a Java style naming convention, not Python.

Critically review the design (copilot)

High Cohesion

Cohesion refers to how closely related the responsibilities of a class are.

- User class: Contains attributes and methods directly related to user identity and actions (e.g. login(), viewScore(), completeQuiz()), which is cohesive. Quiz, Question, and Answer classes: Each focuses on a single concept — managing quizzes, questions, and answers respectively. Their methods (e.g. addQuestion(), editQuestion()) are tightly related to their core purpose.

- Score: Manages scoring logic and data, with methods like calculateScore() and downloadScore() — again, cohesive.
- Event and Chart: These are focused on data visualisation and filtering, which keeps their responsibilities well-defined.
- Conclusion: Most classes have a clear, focused purpose, which supports high cohesion.

✅/⚠️ Low Coupling

Coupling refers to how dependent classes are on each other.

Positive signs:

- Relationships like User → Quiz and Quiz → Question are necessary and logical.
- Use of identifiers (e.g. userId, quizId) in Score helps decouple it from direct object references.
- SystemAdmin inherits from User, which is appropriate given shared attributes and methods.

Areas to watch:

- If Quiz directly holds User objects (createdBy: User), this could increase coupling. Using just userId or a service layer would reduce dependency.
- Chart depends on Event data — this is acceptable, but if Chart directly manipulates Event objects, it could be tightly coupled. A data access layer or API abstraction could help.

Conclusion: Coupling is generally low but could be improved by abstracting direct object references where possible.

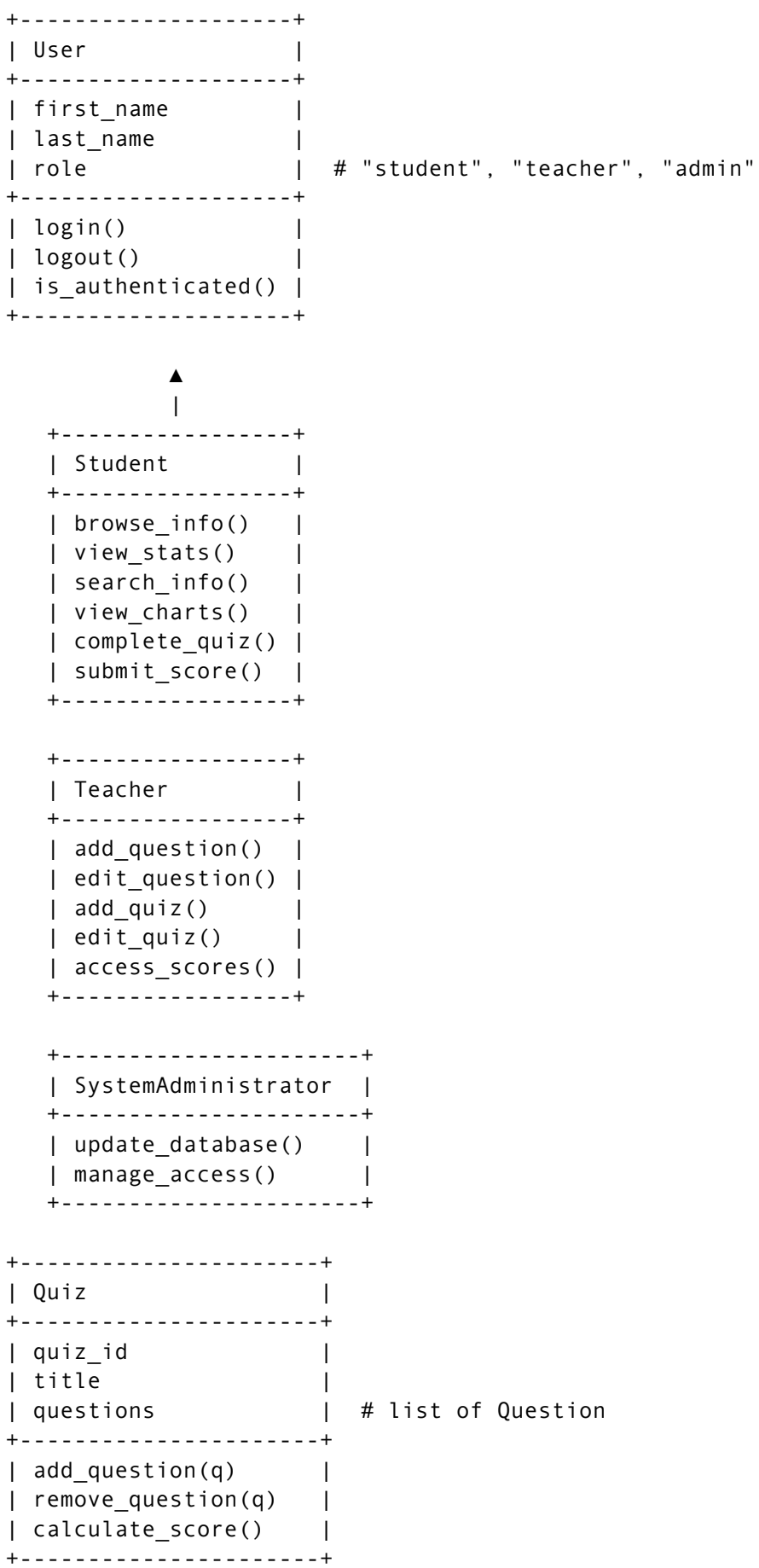
🛠️ Suggestions for Improvement

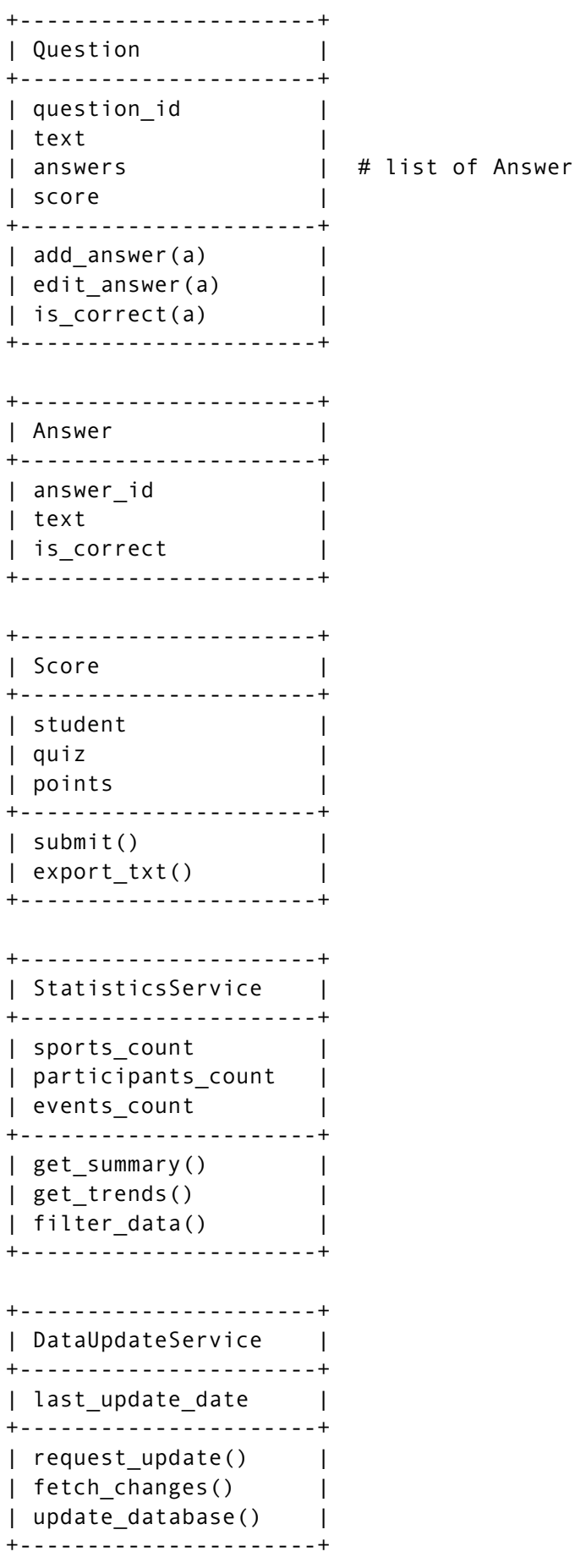
- Introduce interfaces or service classes for operations like authentication, data updates, and chart generation.
- Use dependency injection or data transfer objects (DTOs) to reduce direct dependencies between classes.
- Consider separating data access from business logic to further reduce coupling.

chatgpt response

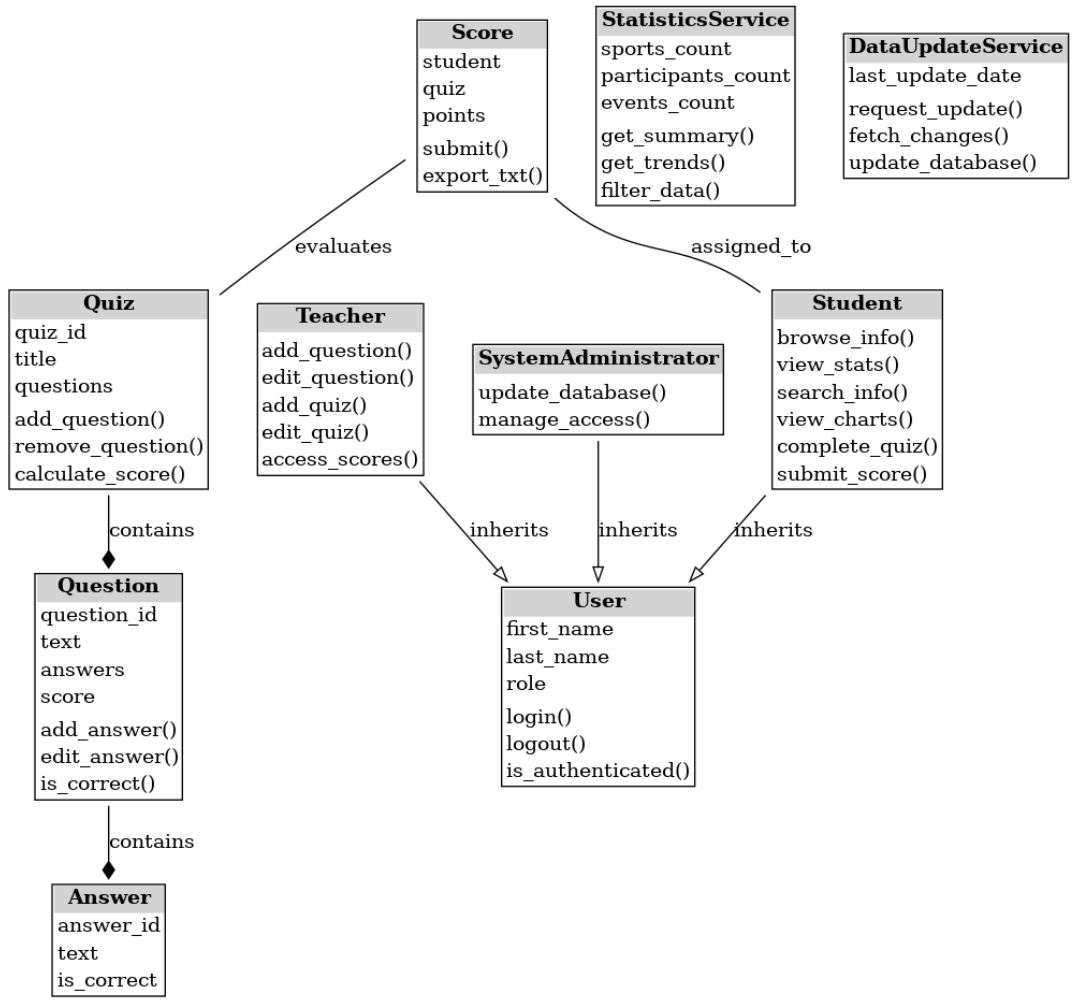
Prompt: `Generate a class diagram from the attached user stories. Include methods. Use Python style naming convention.`

The response was a diagram. Note that this is clearly an AI-generated diagram since a person is unlikely to create a diagram using a cumbersome method of typing `+----` etc. in the terminal.


User
first_name
last_name
role
# "student", "teacher", "admin"
login()
logout()
is_authenticated()
Student
browse_info()
view_stats()
search_info()
view_charts()
complete_quiz()
submit_score()
Teacher
add_question()
edit_question()
add_quiz()
edit_quiz()
access_scores()
SystemAdministrator
update_database()
manage_access()
Quiz
quiz_id
title
questions
# list of Question
add_question(q)
remove_question(q)
calculate_score()


Question
question_id
text
answers
score
# list of Answer
add_answer(a)
edit_answer(a)
is_correct(a)
Answer
answer_id
text
is_correct
Score
student
quiz
points
submit()
export_txt()
StatisticsService
sports_count
participants_count
events_count
get_summary()
get_trends()
filter_data()
DataUpdateService
last_update_date
request_update()
fetch_changes()
update_database()

It also created an image file:



0-1. ChatGPT class diagram

Critically review the design

The response to the prompt `To what extent does this class diagram meet the principles of low coupling and high cohesion?` was:

🔷 Low Coupling

Coupling refers to how dependent classes are on each other.

- Strengths:
  - The diagram separates responsibilities into distinct domains:
    - User hierarchy for authentication and roles.
    - Quiz, Question, Answer, Score for assessment.
    - StatisticsService and DataUpdateService as independent services.
  - Associations use composition/aggregation (e.g. Quiz contains Question, Question contains Answer), which is natural and minimal.
  - Services (StatisticsService, DataUpdateService) are not entangled with User or Quiz objects, reducing dependencies.
- Potential Coupling Issues:
  - Score depends on both Student and Quiz, creating multiple links. This is acceptable since 'score' logically ties both, but it increases coupling.
  - If the statistics service directly queried Quiz or Score objects rather than being an independent data layer, coupling could grow.

✅ Overall: Coupling is relatively low, though further separation of persistence concerns (repositories, data access layer) would reduce dependencies further.

🔷 High Cohesion

Cohesion means each class has a single, well-defined responsibility.

- Strengths:
  - User and its subclasses (Student, Teacher, SystemAdministrator) are cohesive, each handling role-specific actions.
  - Quiz, Question, Answer follow the single responsibility principle: each models one domain concept.
  - StatisticsService only handles data summaries and trends, keeping analytics separate from quiz logic.
  - DataUpdateService only handles API syncing and updates.
- Potential Cohesion Issues:
  - Student has a broad set of methods (browse info, view stats, search info, complete quiz, submit score). This may indicate multiple concerns — browsing/statistics might be better encapsulated in a ParalympicsInfoService instead of inside Student.
  - Teacher has both quiz management and score access. If expanded, score management could be separated into a ScoreService.

✅ Overall: Cohesion is quite high, though Student and Teacher could benefit from delegating responsibilities to service classes for cleaner separation.

⚖️ Summary

- Low Coupling: Achieved by clear separation of users, quizzes, and services. Slightly weakened by Score being tied to both Student and Quiz.
- High Cohesion: Generally good, though Student and Teacher risk becoming "God classes" if more features are added. Extracting browsing/statistics into dedicated services would improve cohesion.

👉 On balance: The diagram meets the principles well, but refinement is possible to further reduce coupling and strengthen cohesion.

Would you like me to refactor the diagram to show an improved version (e.g. moving browsing/stats into a ParalympicsInfoService, and score handling into a ScoreService)?

Modify the design to conform to MVC

Prompt: `Refactor the design to the model view controller design pattern`

Perfect — let's **refactor the design into MVC (Model–View–Controller)**.

The idea is:

- **Model** → domain entities (data & business logic).
- **View** → user-facing components (e.g., web/mobile UI, charts, quiz screens).
- **Controller** → orchestrates between views and models, processes input, enforces rules.

🔷 Refactored MVC Design

Model (Data & Business Logic)

- `User` (abstract), `Student`, `Teacher`, `SystemAdministrator`
- `Quiz`, `Question`, `Answer`, `Score`
- `StatisticsService` (business logic for trends, filtering)

- `DataUpdateService` (synchronisation with API)

View (UI / Presentation Layer)

- `QuizView` → displays quizzes, questions, and answers.
- `ScoreboardView` → shows scores, rankings, leaderboards.
- `StatisticsView` → charts, filters, trends.
- `InfoView` → browsing/searching Paralympics data.

(Views are passive — they render data provided by controllers.)

Controller (Application Logic)

- `AuthController` → login/logout/authentication.
- `QuizController` → manages quiz taking, submission, and grading.
- `TeacherController` → quiz/question editing, score access.
- `StatisticsController` → interacts with `StatisticsService` to provide data to views.
- `SystemAdminController` → handles database updates via `DataUpdateService`.

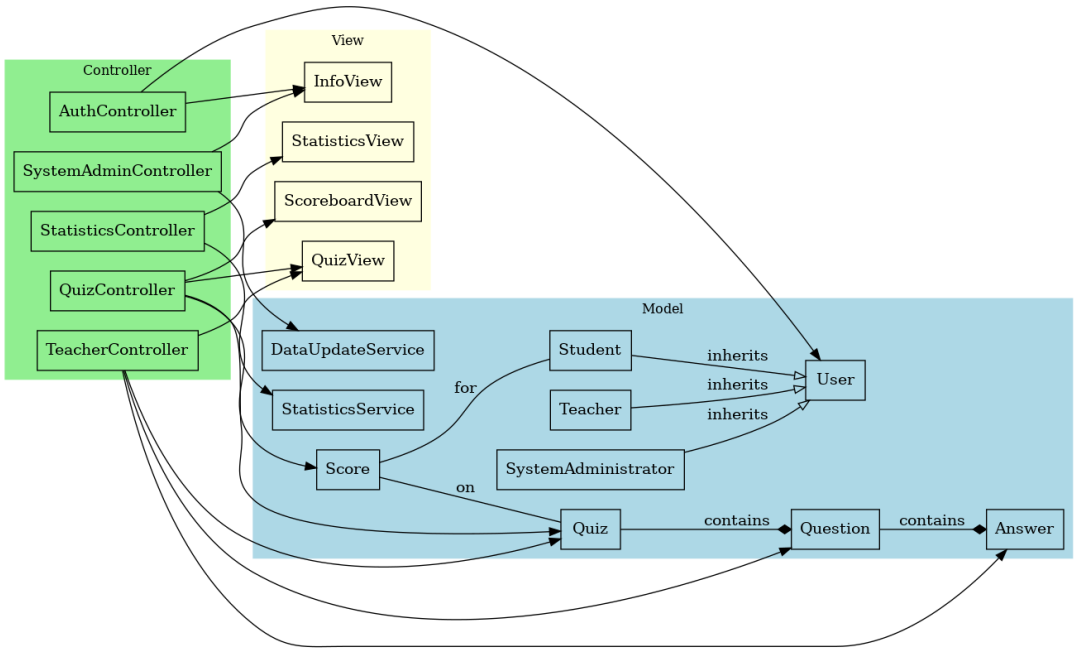
🔷 UML Class Diagram (Textual)

```
              +----------------------+
              |        Model         |
              +----------------------+
User <|-- Student
User <|-- Teacher
User <|-- SystemAdministrator
Quiz *-- Question *-- Answer
Score --> Student
Score --> Quiz
StatisticsService
DataUpdateService

              +----------------------+
              |        View          |
              +----------------------+
QuizView
ScoreboardView
StatisticsView
InfoView

              +----------------------+
              |     Controller       |
              +----------------------+
AuthController --> User
QuizController --> Quiz, Score
TeacherController --> Quiz, Question, Answer
StatisticsController --> StatisticsService
SystemAdminController --> DataUpdateService
```

🔷 Improvements from MVC Alignment

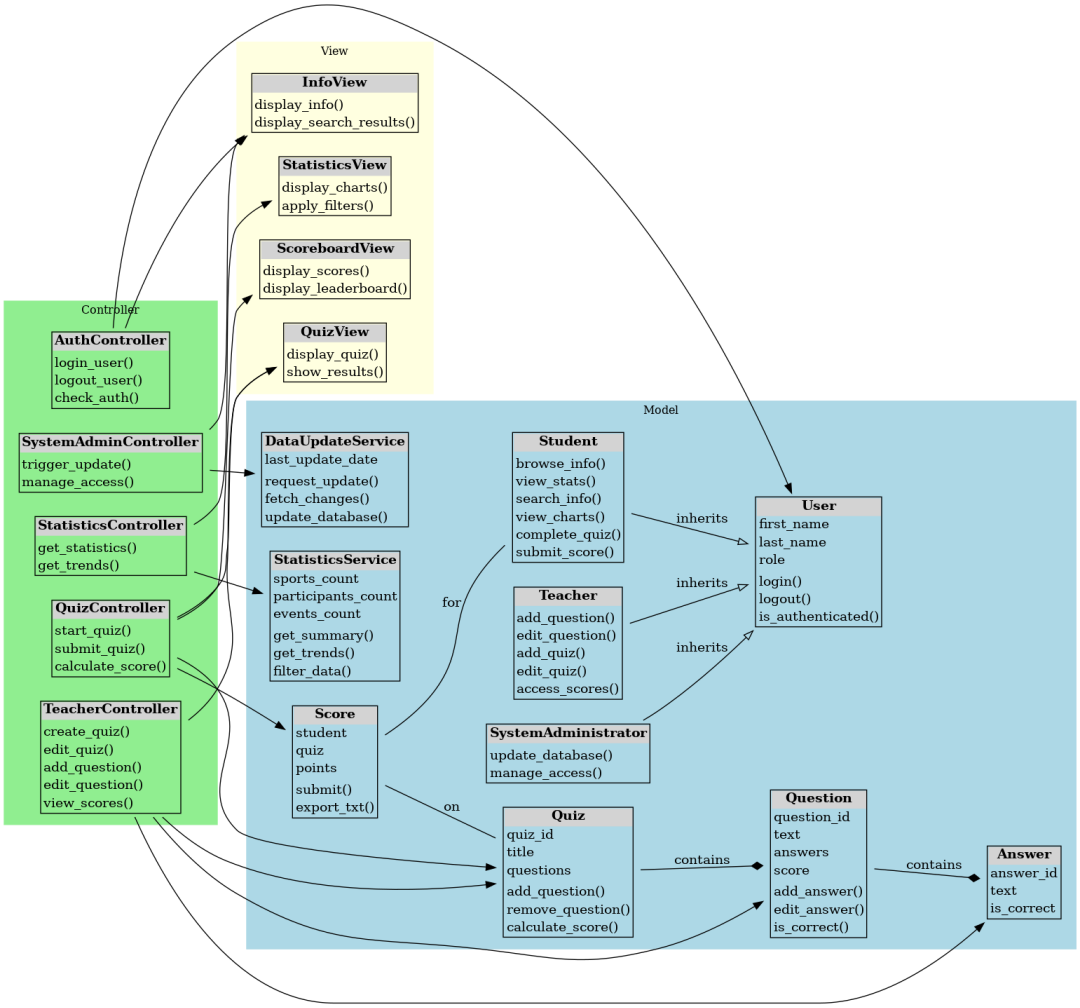
- **Low coupling**:
  - Views do not directly manipulate models — they go through controllers.
  - Services (`StatisticsService`, `DataUpdateService`) remain in the model layer, decoupled from views.
- **High cohesion**:
  - Each controller is focused on a single use case (auth, quiz, teacher, stats, admin).
  - Views only render; models only handle domain logic.

Would you like me to **generate an updated visual UML diagram** (PNG/SVG) with the MVC separation (Models, Views, Controllers grouped) so it's easier to present in documentation?



0-1. MVC class diagram

Subsequent prompt to add attributes and methods:



0-1. MVC class diagram with detail

Appendix

User stories

A prioritised list of user stories used in the AI activity.

Ref	*User story*	*Constraints*	*Priority*
3.	As a student I want to browse information about paralympic games so that I can see what is available.		Must
5.	As a student I want to view statistical information about the paralympics for each event and across events so that I can find information for my project.	Must include number of sports, participants and events.	Must
12.	As a student or teacher I want to be able to access the quiz app from a PC or mobile phone so that I can work at home or in school.		Must
13.	As a student or teacher I want to be able to use the app through a web browser so that I do not have to install an application on my device.	Supports the 3 most frequently used browsers in the UK region.	Must
2.	As a system administrator, I want to make sure that only a teacher can edit questions and quizzes, and access student scores.	Authentication must use the school's single sign on method.	Should
4.	As a student I want to search for information so that I can answer quiz questions.	Search should find data in any of the tables related to the events.	Should
6.	As a student I want to see charts that show trends over time in the events data and be able to refine the charts to specific fields or date ranges so that I can find information for my project.	Support filter by date range, event type (summer/winter), gender and disabilities included. Chart types include line, bar and map.	Should
7.	As a student I want to complete a quiz so that I can assess my learning.		Should

Ref	***User story***	***Constraints***	***Priority***
9.	As a teacher I want to add and edit questions so that these can be added to quizzes.	Allow multiple choice question type.	Should
10.	As a teacher I want to add and edit quizzes, including adding questions to them, so that I can offer my students formative assessment.	A question can be added to more than one quiz. A question has multiple answers. An answer is correct or incorrect. A question has a score.	Should
8.	As a student I want to submit my quiz score so that I can participate in the competition to see who scores the highest.		Could
11.	As a teacher I want to access students' scores to see which student gained the highest score.	Display student first name, last name and score. Optionally allow download to .txt file.	Could
1.	As a system administrator I want to make updates to the paralympic games database so that the data remains accurate and up to date.	Updates will be requested once per day. A request will be triggered in the quiz app and will use the paralympics API to look for changes since the last date of update. Any changes will be updated in the database. The last update date will be stored.	Won't