SWEN90007: Software Design and Architecture Part 2 Report

Team: BrogrammerBrigade

Team Details

Kevin Wu	993272	wukw@student.unimelb.edu.au
James Launder	993934	launderj@student.unimelb.edu.au
Kah Meng Lee	940711	kahl3@student.unimelb.edu.au
Henry Hamer	1173633	hhamer@student.unimelb.edu.au

Implemented Use Cases

1. As a Student, I want to search for upcoming events, so that I can find events hosted by Student Clubs and RSVP to them.

2. As a Student, I want to RSVP to an event, so that I can attend events hosted by Student Clubs.
3. As a Student, I want to create an event for a Student Club I administer, so that I can organise activities and gatherings.
4. As a Student, I want to cancel an event for a Student Club I administer, so that I can remove events as needed.
5. As an administrator of a Student Club, I want to add other Students as administrators, so that they can create, amend or cancel events as needed.
6. As an administrator of a Student Club, I want to remove other Students as administrators, so that they can no longer create, amend or cancel events as needed.
7. As an administrator of a Student Club, I want to create an application for funding, so that the Student Club can hold event/s.
8. As a Student, I want to cancel one or more RSVPs to an event, so that it shows the correct number of students attending.
9. As an administrative member of a Student Club, I want to modify an event within my Student Club, so that events contain accurate information.

Implemented Patterns

- Domain model
- Data mapper
- Unit of work
- Lazy load
- Identity field
- Foreign key mapping
- Association table mapping
- Concrete Table inheritance
- Authentication and Authorization

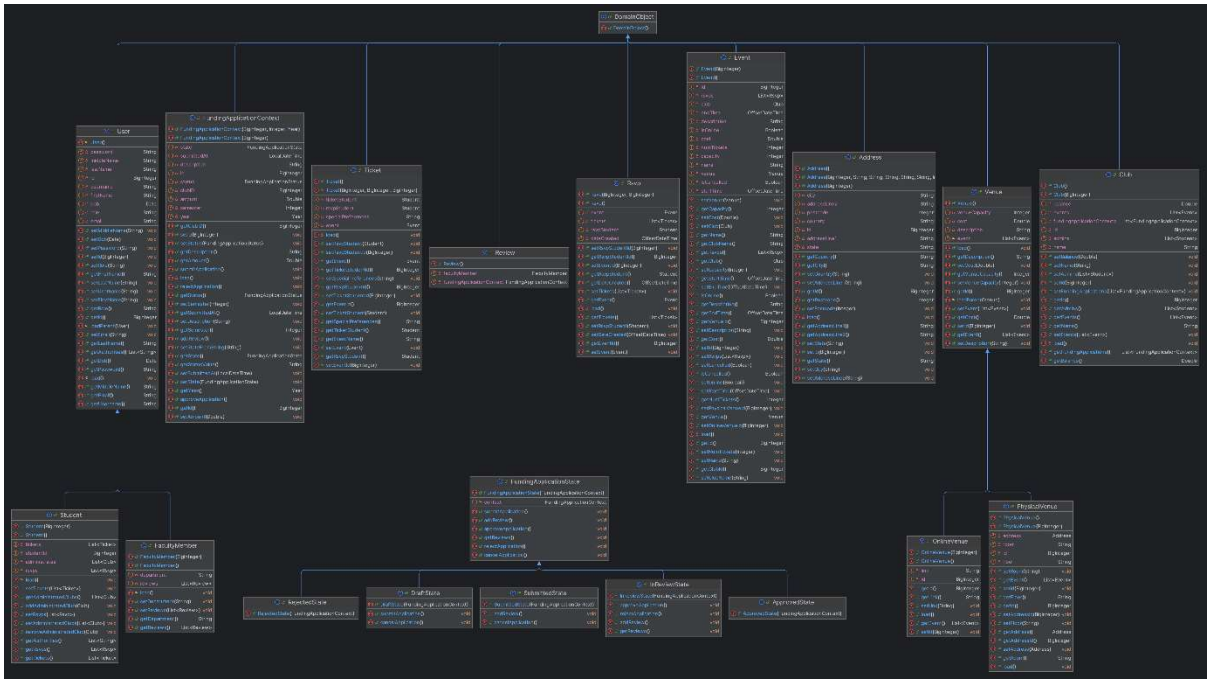
Class Diagrams

Our project consists of five main layers: controller, service, domain, mapper and database connection. For clarity, we decided that it would be best to split up the class diagram as follows, since the class diagram for the entire system is too large and complex to be realistically interpreted.

Domain Layer Diagram

This diagram shows the domain classes in our implementation. As shown there are two main hierarchies, the main domain object hierarchy as well as the fundingApplication state hierarchy which is used for the Status pattern described below.

For better viewing: https://github.com/feit-swen90007-2024/BrogrammerBrigade/blob/1f6963b6b258264bc6c55141ca51981144fb7726/diagrams/uml_part2/domain.png



Controller -> Service Layer Diagram

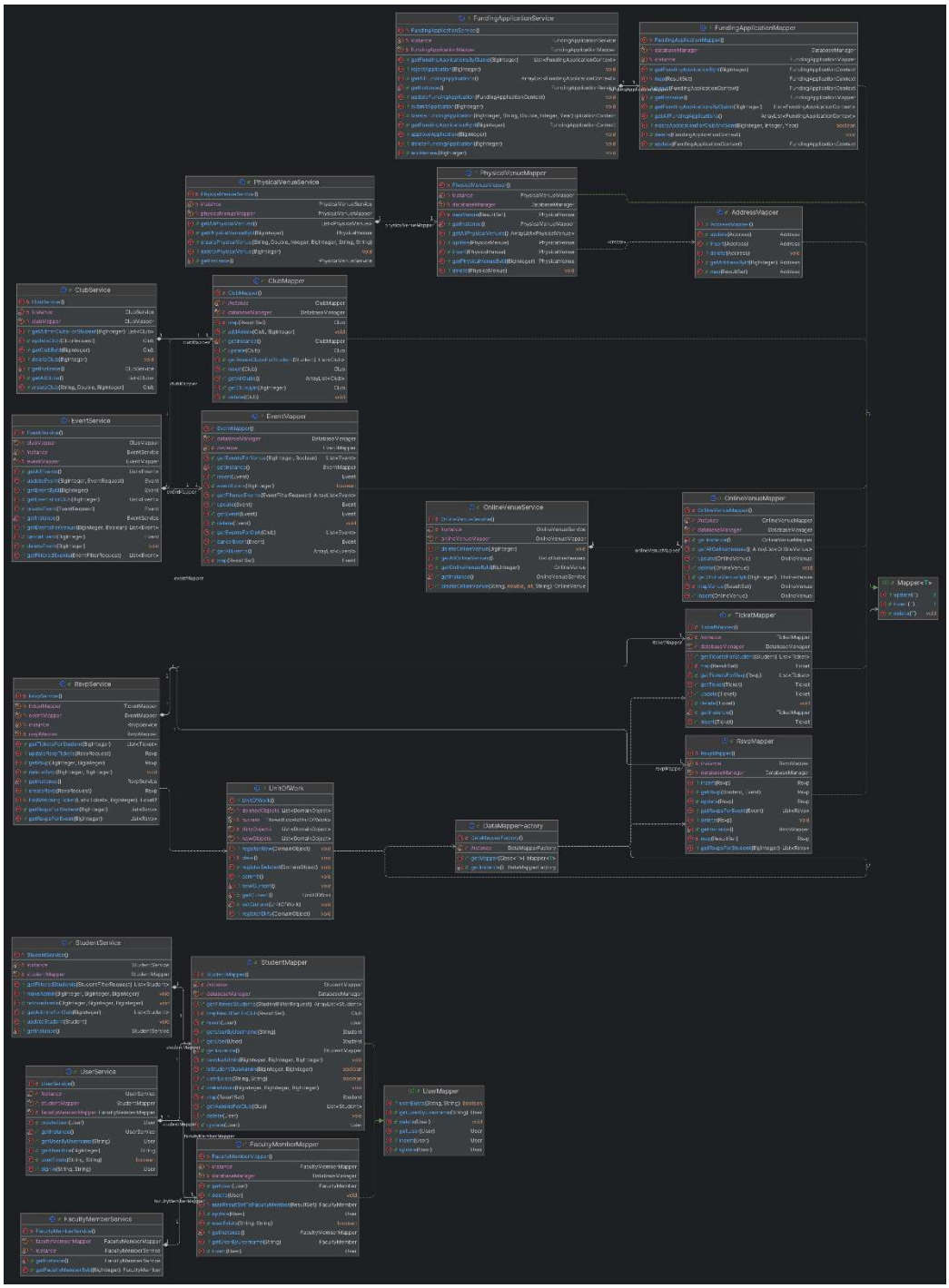
This diagram shows dependencies between our controller and service layers. As shown, there is a clear separation of concerns here with each controller interacting only with its associated service class/s.

For better viewing: https://github.com/feit-swen90007-2024/BrogrammerBrigade/blob/1f6963b6b258264bc6c55141ca51981144fb7726/diagrams/uml_part2/controllerService.png

Service -> Mapper Layer Diagram

The below diagram shows dependencies between the service layer and mapper layer classes. This interaction is a little more complex, since multiple mapper objects are often needed in the service classes to retrieve or update more than one different domain class from the database. For example, the event mapper must interact with the rsvp class to keep track of the number of RSVP tickets for an event.

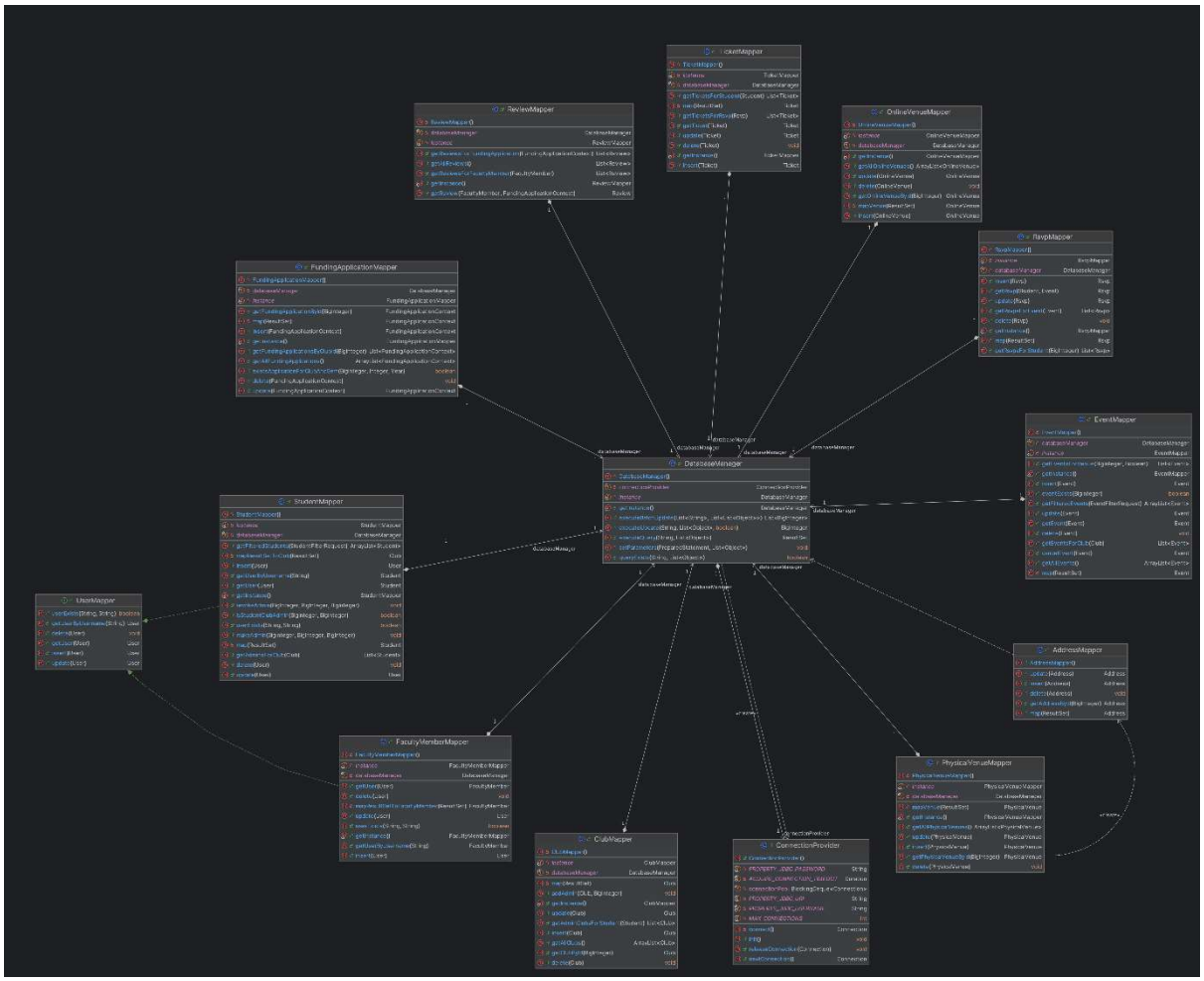
For better viewing: https://github.com/feit-swen90007-2024/BrogrammerBrigade/blob/1f6963b6b258264bc6c55141ca51981144fb7726/diagrams/uml_part2/serviceMapper.png



Mapper -> Database Connection Layer Diagram

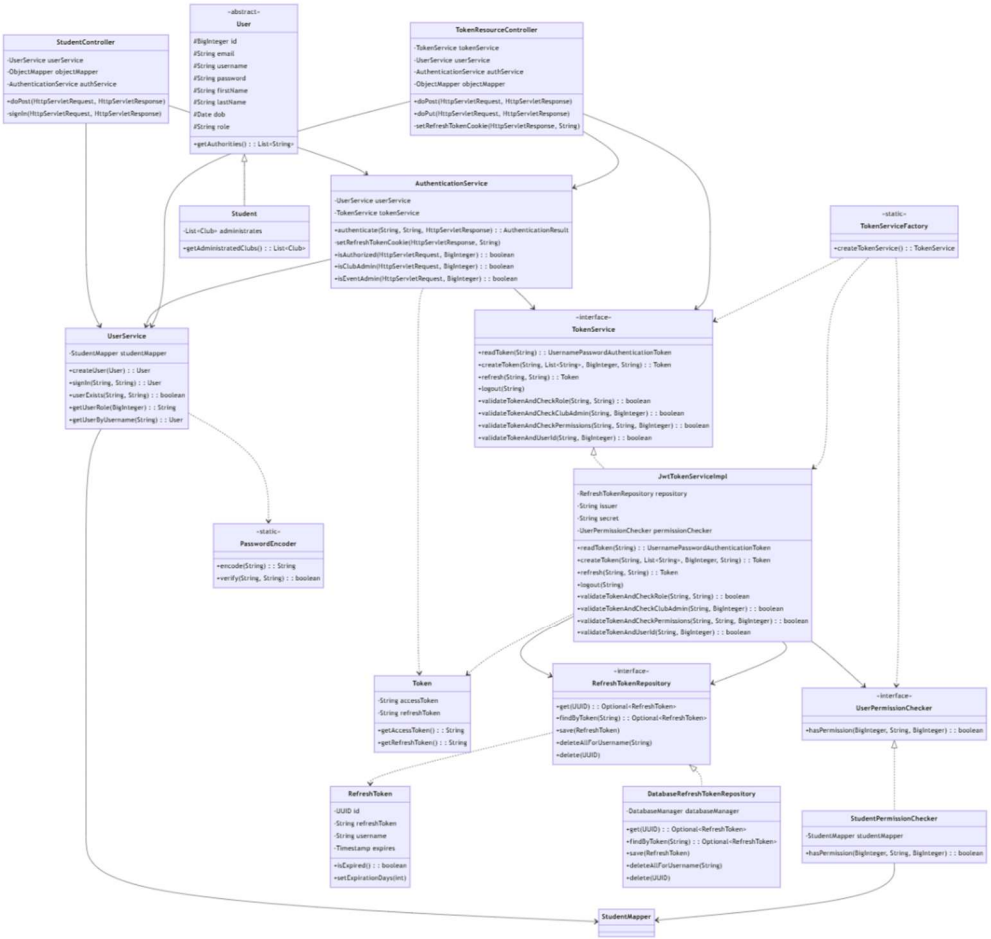
This diagram shows associations between classes in the mapper and DB connection layer, which acts as an adapter to the database to decouple query creation (mapper) and DB connection logic. As shown, all mappers send queries and recieve results from the DataBaseManager object, which in turn uses the ConnectionProvider object to handle connections.

Better viewing: https://github.com/feit-swen90007-2024/BrogrammerBrigade/blob/1f6963b6b258264bc6c55141ca51981144fb7726/diagrams/uml_part2/mapperDb.png

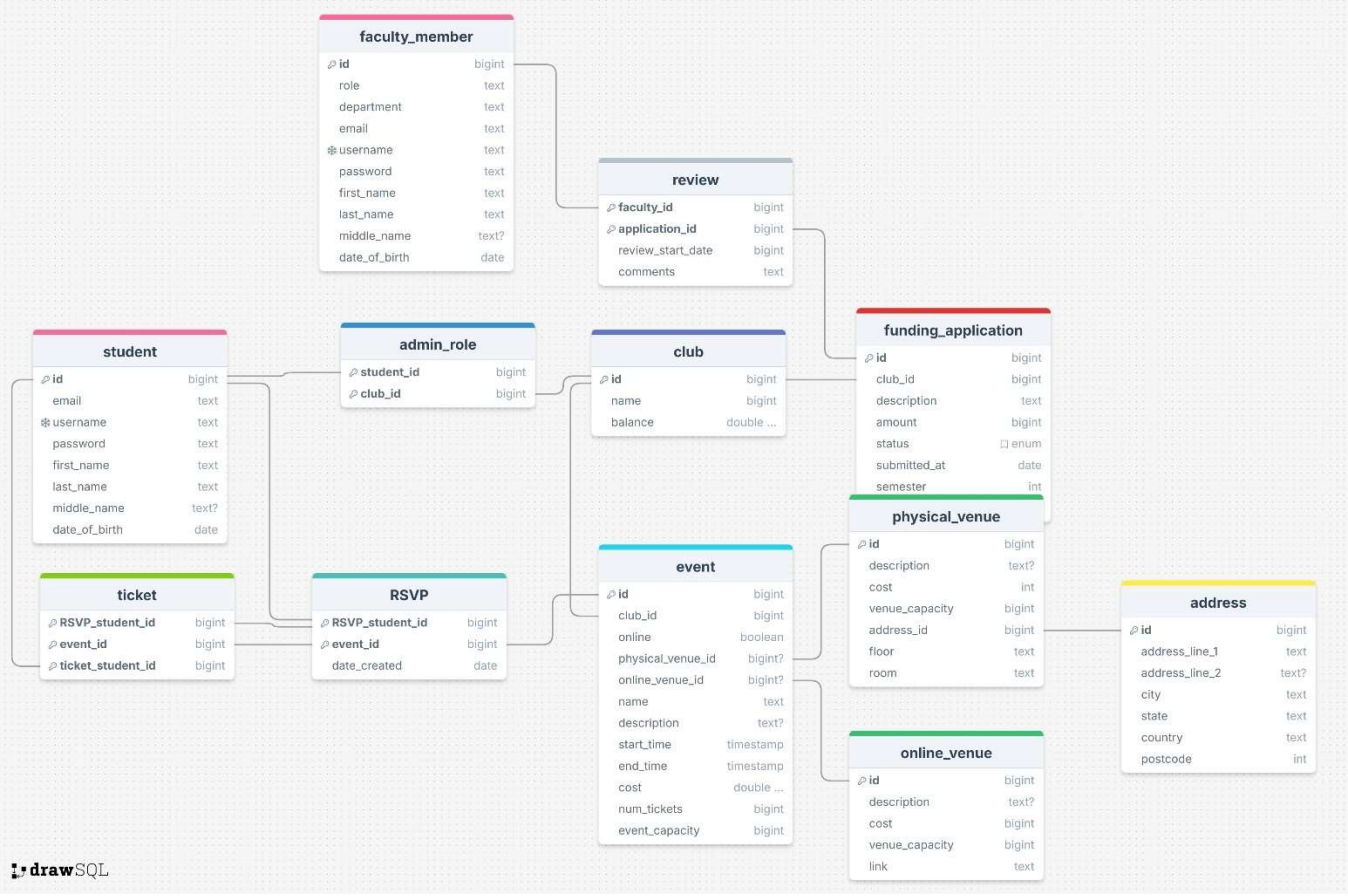


Authentication

This diagram shows associations between classes in the authentication and token management layer, which acts as protection layer for user access and session handling. As shown, the controllers interact with the AuthenticationService, which coordinates authentication using the UserService and TokenService. The TokenService, implemented by JwtTokenServiceImpl, manages token operations and interacts with the RefreshTokenRepository for token persistence and session management. The UserService utilizes the StudentMapper for database operations like getting the clubs a student is admin of and PasswordEncoder for using Bccrypt to protect password. The StudentPermissionChecker verifies specific permissions for students.


StudentController
-UserService userService
-ObjectMapper objectMapper
-AuthenticationService authService
+doPost(HttpServletRequest, HttpServletResponse)
-signIn(HttpServletRequest, HttpServletResponse)
~abstract~
User
#BigInteger id
#String email
#String username
#String password
#String firstName
#String lastName
#Date dob
#String role
+getAuthorities() : : List<String>
TokenResourceController
-TokenService tokenService
-UserService userService
-AuthenticationService authService
-ObjectMapper objectMapper
+doPost(HttpServletRequest, HttpServletResponse)
+doPut(HttpServletRequest, HttpServletResponse)
-setRefreshTokenCookie(HttpServletResponse, String)
Student
-List<Club> administrates
+getAdministratedClubs() : : List<Club>
AuthenticationService
-UserService userService
-TokenService tokenService
+authenticate(String, String, HttpServletResponse) : : AuthenticationResult
-setRefreshTokenCookie(HttpServletResponse, String)
+isAuthorized(HttpServletRequest, BigInteger) : : boolean
+isClubAdmin(HttpServletRequest, BigInteger) : : boolean
+isEventAdmin(HttpServletRequest, BigInteger) : : boolean
~static~
TokenServiceFactory
+createTokenService() : : TokenService
UserService
-StudentMapper studentMapper
+createUser(User) : : User
+signIn(String, String) : : User
+userExists(String, String) : : boolean
+getUserRole(BigInteger) : : String
+getUserByUsername(String) : : User
~interface~
TokenService
+readToken(String) : : UsernamePasswordAuthenticationToken
+createToken(String, List<String>, BigInteger, String) : : Token
+refresh(String, String) : : Token
+logout(String)
+validateTokenAndCheckRole(String, String) : : boolean
+validateTokenAndCheckClubAdmin(String, BigInteger) : : boolean
+validateTokenAndCheckPermissions(String, String, BigInteger) : : boolean
+validateTokenAndUserId(String, BigInteger) : : boolean
~static~
PasswordEncoder
+encoder(String) : : String
+verify(String, String) : : boolean
JwtTokenServiceImpl
-RefreshTokenRepository repository
-String issuer
-String secret
-UserPermissionChecker permissionChecker
+readToken(String) : : UsernamePasswordAuthenticationToken
+createToken(String, List<String>, BigInteger, String) : : Token
+refresh(String, String) : : Token
+logout(String)
+validateTokenAndCheckRole(String, String) : : boolean
+validateTokenAndCheckClubAdmin(String, BigInteger) : : boolean
+validateTokenAndCheckPermissions(String, String, BigInteger) : : boolean
+validateTokenAndUserId(String, BigInteger) : : boolean
Token
-String accessToken
-String refreshToken
+getAccessToken() : : String
+getRefreshToken() : : String
~interface~
RefreshTokenRepository
+get(UUID) : : Optional<RefreshToken>
+findByToken(String) : : Optional<RefreshToken>
+save(RefreshToken)
+deleteAllForUsername(String)
+delete(UUID)
~interface~
UserPermissionChecker
+hasPermission(BigInteger, String, BigInteger) : : boolean
RefreshToken
-UUID id
-String refreshToken
-String username
-Timestamp expires
+isExpired() : : boolean
+setExpirationDays(int)
DatabaseRefreshTokenRepository
-DatabaseManager databaseManager
+get(UUID) : : Optional<RefreshToken>
+findByToken(String) : : Optional<RefreshToken>
+save(RefreshToken)
+deleteAllForUsername(String)
+delete(UUID)
StudentPermissionChecker
-StudentMapper studentMapper
+hasPermission(BigInteger, String, BigInteger) : : boolean
StudentMapper

Database Schema Diagram



Patterns and Descriptions

ADR - Domain Model Pattern Implementation

Context

Our application will need to be able to handle complex relationships between entities such as students, faculty members, clubs, event etc. It must also be capable of efficiently processing a high volume of transactions and complex queries. Additionally, the system should be flexible enough to accommodate changes in university policies and procedures over time, as well as potential integration with other university systems in the future.

Decision

We have implemented the pattern for all domain objects represented in the class diagram.

Implementation Strategy

- Create an object model that represents the complex relationships between entities.
- Encapsulate behaviour associated with each entity within its respective domain object.
- Design the domain model to allow for easy extension and modification of business rules.

We decided on the following domain entities to model our system:

- Student and Faculty Member, who inherit from User: These represent the two user classes of the system.
- Club: This represents a student club.
- Event: This represents an event honest by a club.
- Online Venue and Physical Venue, which inherit from Venue: An online or physical location and time where an event is hosted.
- Address: Stores an address for a venue object.
- Review: Stores comments and a review decision by a faculty member for a funding application.
- Funding Application: Represents an application for funding by a student club.
- RSVP: Represents an RSVP made by a student to the event
- RSVP Ticket: Represents the ticket associated to the RSVP. A single RSVP can only have 3 tickets.

All of these classes inherit from a parent DomainObject class.

Consequences

Positive

- The Domain Model pattern provides a structure that can scale well with increasing complexity, making it suitable for handling a high volume of transactions and complex queries efficiently.
- It offers greater flexibility for incorporating new rules or modifying existing ones as university policies and procedures change over time.
- It allows for future extensions to be implemented easily, allowing new entities or relationships to be added to the existing model without major restructuring.

Negative

- The Domain Model pattern may introduce higher initial complexity compared to simpler patterns like Transaction Script or Table Module.

Compliance

- All features and entities must be modelled using the domain model pattern

Alternatives Considered

Alternatives such as Table module and Transaction script were considered, but ultimately not chosen due to issues with extensibility and maintainability.

Notes

Author	Version	Changelog
Kah Meng	1.0	Initial Proposed version

ADR- Data Mapper

Context

Our application needs to handle retrieval and data manipulation operations. Our initial approach from part 1A combines all endpoint code onto the controller. This is risky as it tightly couples business logic with data access code. To address this, we need a pattern that provides a clear separation of concerns between the domain logic and database interactions.

Decision

We have implemented this pattern for all domain objects in the domain diagram. So, each domain object will have a controller and a mapper. The concept of layering has been expanded upon by our team. This additional layering of our application is discussed under ADR – Layering.

Implementation Strategy

- Introduce a Data Access Layer that contains mapper classes for each domain object such as Student.
- These mapper classes will be responsible for transferring data between their corresponding database tables and domain objects.
- Abstract database code away from the controllers and encapsulate it in the data mapper classes.

Consequences

Positive

- The Data Mapper pattern allows us to achieve higher cohesion by separating database interactions from the controller logic. Controllers can focus on handling data from requests, while mappers are responsible for SQL code and database interactions.
- Due to the higher cohesion, maintenance becomes easier. For example, if there's a SQLException, changes are isolated to the Mapper classes.
- Changes to the database schema or switching to a different database system would primarily affect only the Data Mapper layer, not the entire application.

Negative

- Introducing a separate layer for data mapping adds some complexity to the overall system architecture.
- Initially, it may take more time to develop separate mapper classes for each domain object. Moreover, if we introduce a new domain object to our application, there would be more code to write compared to not implementing it.

Compliance

- All controllers should have their own mappers.
- All database interactions must go through the service layer which then calls the mapper. More details regarding this abstraction can be found in ADR – layering.
- Controllers should not contain direct SQL queries or database connection code.

Alternatives Considered - Active Record Pattern

Consequences

While the Active Record pattern was implemented at the start, it was ultimately decided that it wouldn't provide the level of separation between domain logic and data access that we require for this system moving forward.

Positive

- The biggest strength is its simplicity to implement.

Negative

- It tightly couples domain logic with database access code, which would make future extensions difficult.
- If there are SQL exceptions, we would need to make changes to a class that has domain layer logic. Explicit separation of the layers would allow us to maintain the system more efficiently.
- We cannot reuse identical database access code from other controllers. With the Data Mapper pattern, we will be able to easily access database code from other controllers by calling their respective mappers.

Author: Kah Meng

Changelog

Version	Changes
V1.0	Initial implementation of Data Mapper, moving away from Transaction scripts.
V1.1	Decision to split the application into further layers. So, the Data Mapper's responsibility will be split into multiple classes. Further details on layering will be under ADR layered architecture.

ADR – Layered Architecture

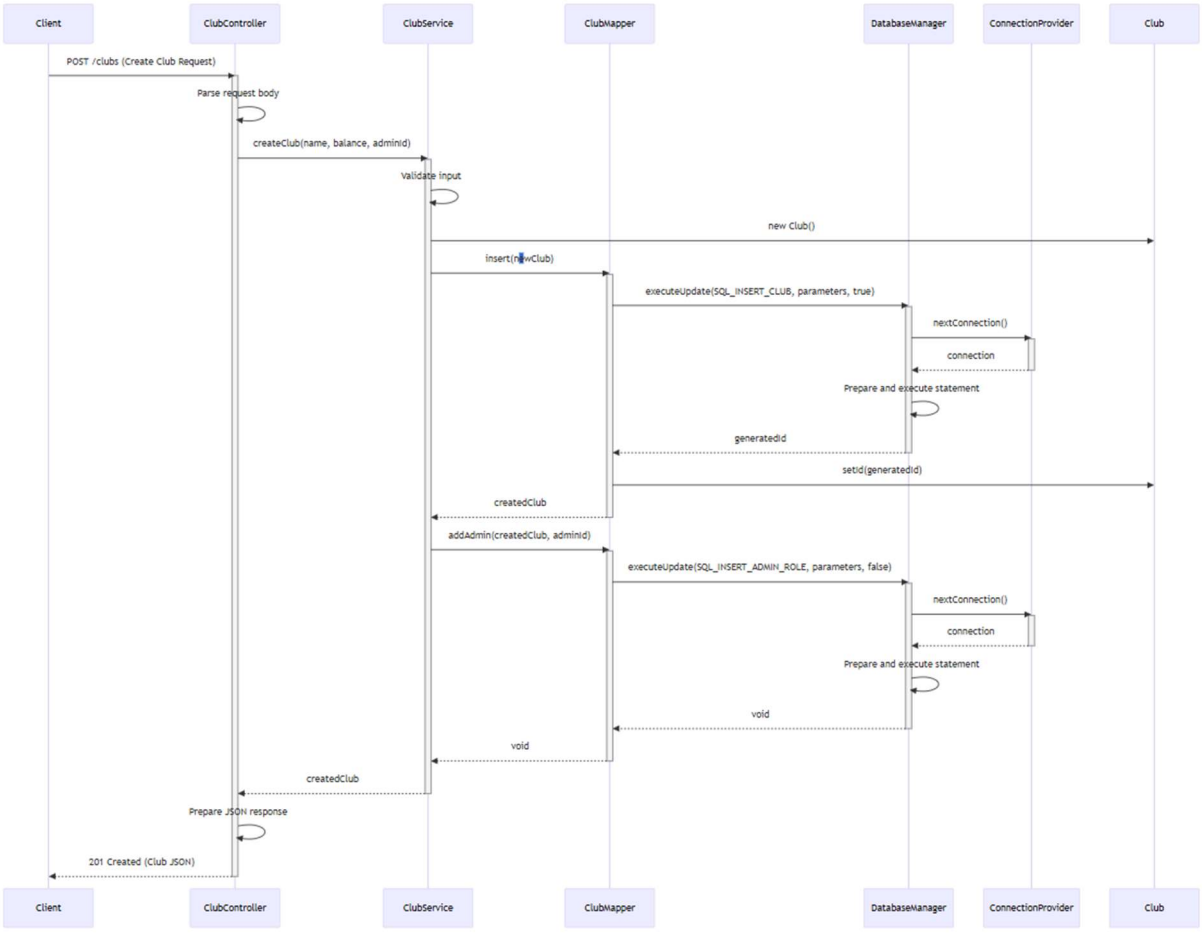
Context

Our backend is built using a layered architecture, separating concerns across distinct layers: controller, service, domain, mapper, and database. Each layer has a specific responsibility to ensure modularity, maintainability, and ease of development, described in more detail below.

Decision

We will continue to implement and maintain a layered architecture across the Java backend to promote separation of concerns and improve scalability and testability. Each domain class will have its corresponding controller, service, and mapper classes, with the database layer abstracted into manager and connection classes.

Implementation Strategy



- **Controller Layer**: Each controller class handles specific HTTP requests for a particular domain object (e.g., StudentController, EventController). It converts incoming requests into DTOs or individual parameters and invokes the service layer.
- **Service Layer**: Corresponding service classes (e.g., StudentService, EventService) handle business logic and mediate between the controller and domain layers. Service classes call methods on domain objects, orchestrating behavior based on incoming requests.

- **Domain Layer**: Domain classes encapsulate business logic related to each entity (e.g., Student, Event). They contain attributes and methods but are decoupled from persistence logic.
- **Mapper Layer**: Mapper classes (e.g., StudentMapper, EventMapper) translate domain objects into SQL queries. They interact with the database layer to persist or retrieve data. Mappers communicate directly with the database manager to execute SQL statements.
- **Database Layer**: The DatabaseManager and ConnectionProvider manage the connection pool, handle authentication, and provide methods for executing SQL queries. All database transactions are handled through this layer.

It should also be noted that all classes, except those in the domain layer, are implemented as singletons within their thread to ensure that only one instance is created per request. This reduces memory usage and avoids redundant object creation, while also ensuring consistency in managing shared resources, such as database connections, across the application.

Consequences

Positive

- Clear separation of responsibilities improves code organization, making the system easier to maintain and extend.
- Enforcing strict input/output types in each layer ensures consistency in data flow and allows for compile-time type checking, reducing the risk of type-related errors. This pattern promotes clear separation of concerns, making the system more reliable and easier to maintain.

Negative

- Introducing multiple layers increases the overall complexity of the system. Developers need to ensure that communication between layers is well-managed.
- Passing data across layers adds some performance overhead. However, the benefits of modularity and maintainability outweigh this cost in most cases.
- With each domain class having a controller, service, and mapper class, there is an increased maintenance burden to keep all layers synchronized.

Compliance

All current and future domain classes will follow this layered architecture. Each domain entity should ideally have corresponding controller, service, and mapper classes. There is some grey area here for objects that are simple and deeply coupled with other objects (e.g. for tickets and rsvp a shared controller might be considered) but in general each concrete domain entity (this excludes purely relational database entities like admin roles) should follow this. The database layer will continue to be managed via the DatabaseManager and ConnectionProvider classes.

Considered Alternative 1 – Combined Layers

We considered combining the service and controller layers to reduce complexity. This would mean that the controller would also handle business logic, reducing the number of classes. However, we opted against this approach as it would violate the principle of separation of concerns, making testing and code maintenance more difficult.

Consequences

- **Positive:** Combining layers would reduce the number of classes and simplify communication between the controller and service logic.
- **Negative:** It would result in tightly coupled code, making testing and future extension harder. This approach would also reduce flexibility, as the controller would be responsible for both request handling and business logic.

Author: Henry Hamer

Changelog

Version	Changes
V1.0	Initial implementation of layered architecture decision. The architecture separates the controller, service, domain, mapper, and database layers.
V1.1	Introduced type enforcement across layers to ensure consistency and correctness:

ADR - Unit of Work Pattern for RSVP Ticket management

Context

Students may need to manage multiple tickets related to an RSVP in one transaction. These tickets can be added, updated, or deleted in a single front-end form submission. The complexity arises from ensuring that only the modified tickets are updated, while unmodified ones remain unchanged. Without an efficient mechanism, unnecessary database updates could result in performance issues. To address this, the Unit of Work pattern was implemented, allowing for more efficient database transactions.

Decision

We will implement the Unit of Work pattern in the RSVP ticket management feature to handle efficient updates and management of database operations.

Implementation Strategy

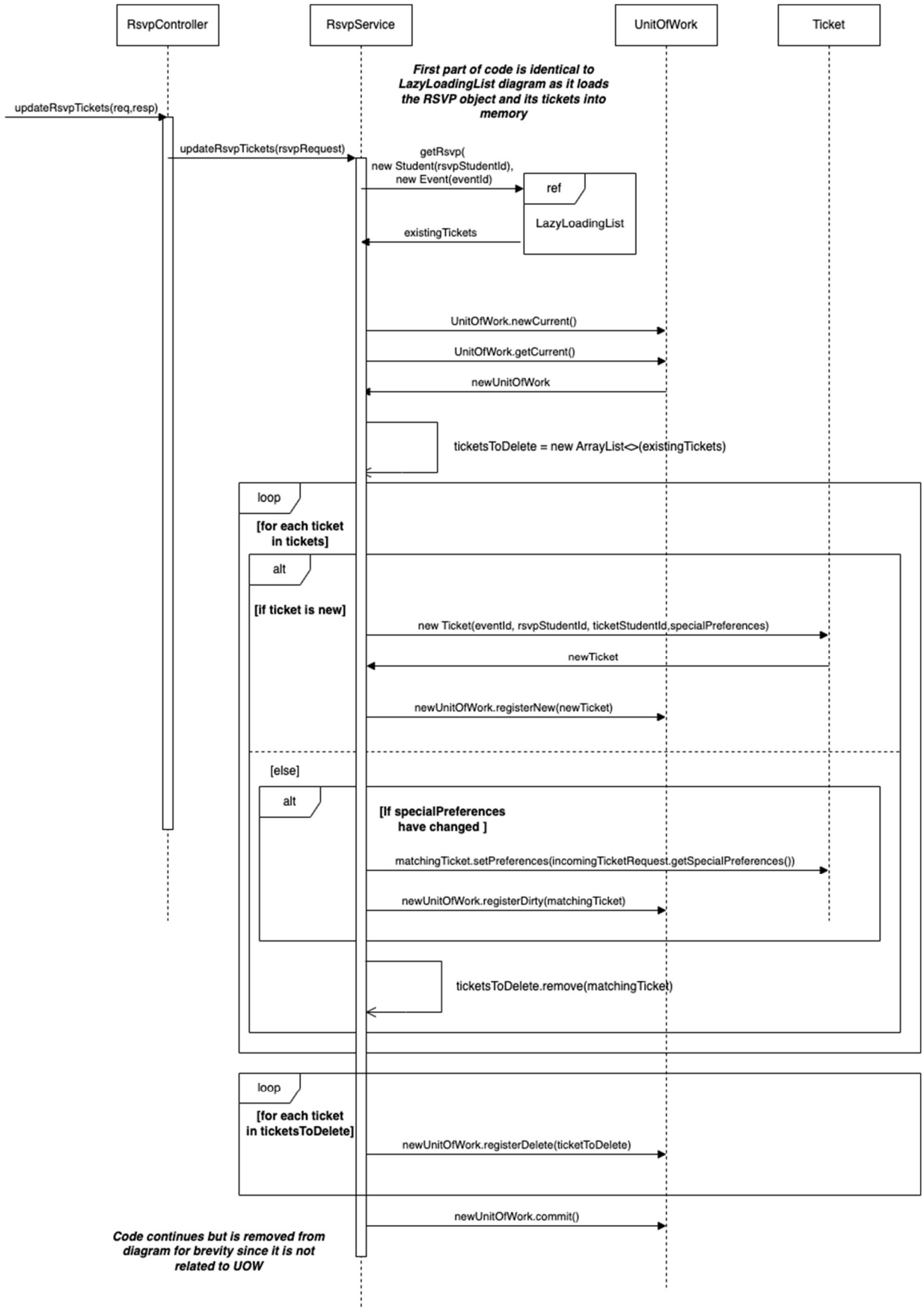
The Unit of Work class is called from the service layer. We chose to implement the pattern using caller registration, allowing the service layer to explicitly register tickets that need to be added, updated, or deleted. This ensures flexibility in determining which tickets are involved in the operation, making it easier to implement business rules before registration.

The implementation revolves around maintaining lists for new, dirty, and deleted objects, and persisting changes in a single transaction:

- **Service Layer Integration:** In the RsvpService class, the updateRsvpTickets method is responsible for handling updates to the tickets. A UnitOfWork instance is created at the start of the method to track changes across the operation.
- **Selective Object Registration**: By comparing tickets submitted from the frontend form with the ticket list from the database, tickets are registered with registerNew() for newly added tickets, registerDirty() for tickets with modifications (such as changed dietary preferences), and registerDelete() for tickets to be deleted.

- **Committing Changes:** Once all changes are registered, the commit() method is called, executing all changes in a single transaction.

For more details, see the sequence diagram below.


RsvpController
RsvpService
UnitOfWork
Ticket
First part of code is identical to LazyLoadingList diagram as it loads the RSVP object and its tickets into memory
updateRsvpTickets(req,resp)
updateRsvpTickets(rsvpRequest)
getRsvp(
new Student(rsvpStudentId),
new Event(eventId)
ref
LazyLoadingList
existingTickets
UnitOfWork.newCurrent()
UnitOfWork.getCurrent()
newUnitOfWork
ticketsToDelete = new ArrayList<>(existingTickets)
loop
[for each ticket in tickets]
alt
[if ticket is new]
new Ticket(eventId, rsvpStudentId, ticketStudentId,specialPreferences)
newTicket
newUnitOfWork.registerNew(newTicket)
[else]
alt
[If specialPreferences have changed]
matchingTicket.setPreferences(incomingTicketRequest.getSpecialPreferences())
newUnitOfWork.registerDirty(matchingTicket)
ticketsToDelete.remove(matchingTicket)
loop
[for each ticket in ticketsToDelete]
newUnitOfWork.registerDelete(ticketToDelete)
newUnitOfWork.commit()
Code continues but is removed from diagram for brevity since it is not related to UOW

Consequences

Positive

- The Unit of Work pattern allows the system to avoid unnecessary updates to unchanged tickets, which increases the performance of database transactions. Only the modified (dirty) tickets are updated, and this can be applied to other use cases involving partial updates.
- By coordinating changes across multiple related objects, Unit of Work ensures that updates to an RSVP and its related tickets (which are associated with students and events) are handled consistently in one operation.

Negative

- While Unit of Work adds flexibility, it introduces additional code and complexity, requiring careful coordination in the logic to avoid erroneous registrations and maintain consistency.

Compliance

Any changes to tickets in the RSVP system must now be registered using the Unit of Work pattern. Future use cases involving similar partial updates across object lists may also adopt the same pattern.

Considered Alternative 1 – Object Registration

Instead of using caller registration, we considered object registration, where the domain object itself registers changes. However, we opted against this approach to keep domain objects free of persistence logic. Caller registration provides more flexibility and better control over object registration.

Consequences

- **Positive:** Object registration simplifies the process by allowing domain objects to handle their own persistence state, reducing the need for external tracking and lowering the amount of boilerplate code in the service layer. This encapsulation improves the cohesion of the system, making it easier to manage individual objects.
- **Negative**: The approach limits flexibility since business logic and conditions become harder to apply before registration. It also introduces tighter coupling between domain logic and persistence, making objects harder to maintain and violating the separation of concerns principle.

Considered Alternative 2 - No Unit of Work Pattern

We considered not implementing the Unit of Work pattern at all and simply handling each ticket update individually as it comes. This alternative would result in separate transactions for each addition, update, or deletion. Although simpler to implement, this approach would have led to a higher likelihood of performance degradation, increased database overhead, and the risk of inconsistencies if one part of the operation failed.

Consequences

- **Positive**: Less implementation complexity and lower initial development time.

- **Negative**: No optimization for batch updates, increased performance issues over time, and a higher risk of inconsistent database states during operations that modify multiple tickets.

Author: Henry Hamer

Changelog:

Version	Changes
V1.0	Designed Initial unit of work implementation using object registration
V1.1	Updated design to use caller registration after considering above consequences and realising that we would not be implementing unit of work universally for all database transactions.

ADR – Hybrid Lazy Loading Pattern

Context

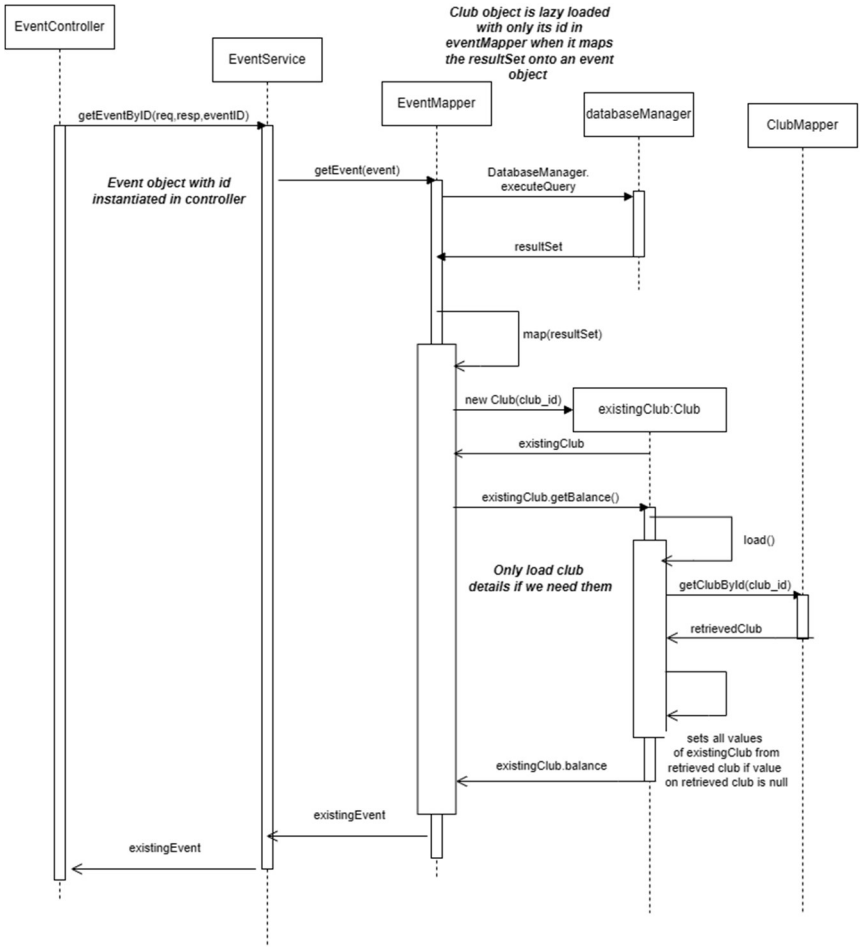
As our application needs to handle a large volume of data efficiently. We need a strategy to optimize data retrieval and memory usage, especially when dealing with complex relationships between entities.

Decision

We have decided to implement a hybrid lazy loading strategy, combining ghost loading and lazy initialization across all domain objects.

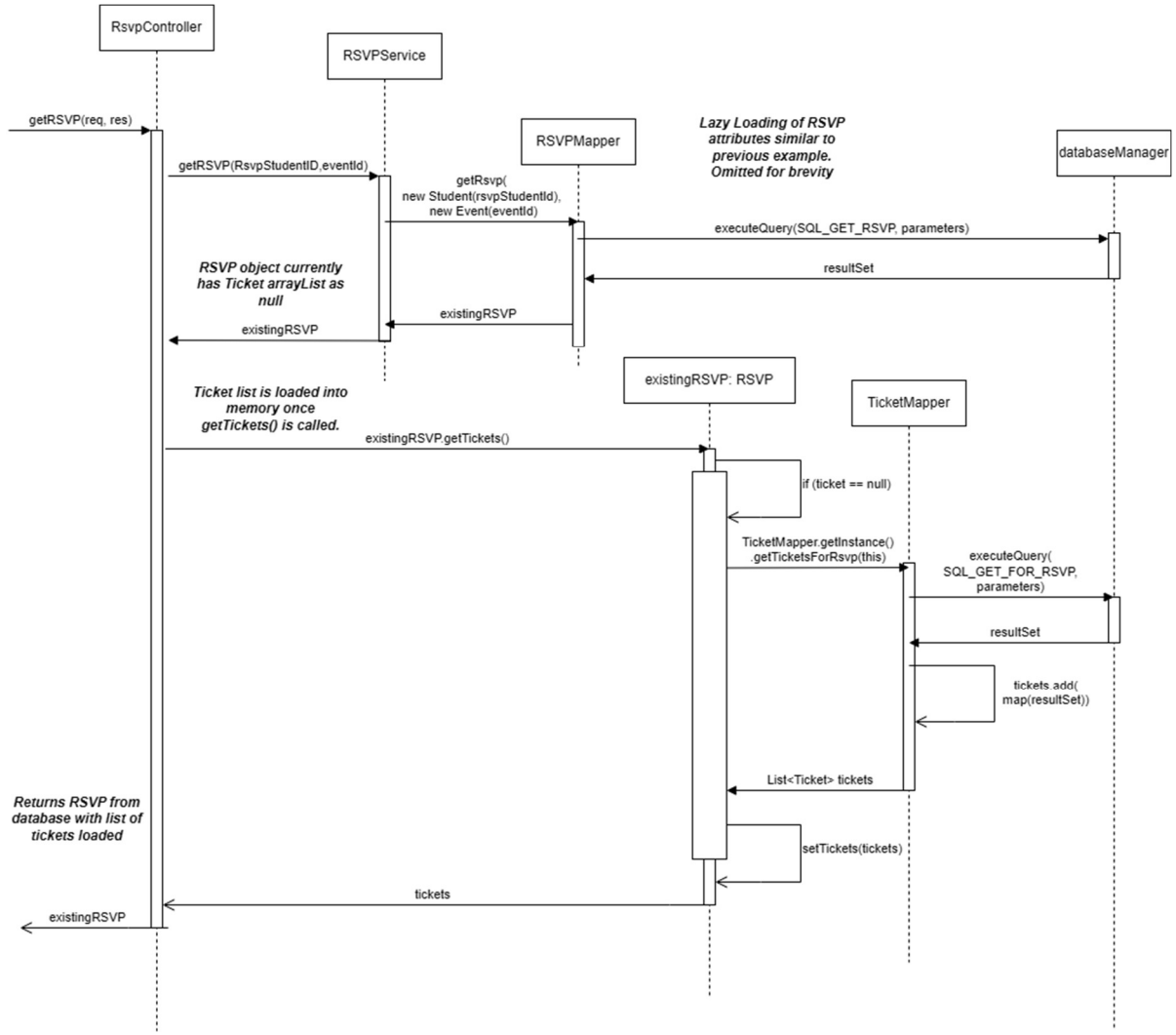
Implementation Strategy

Ghost



We will apply ghost loading to attributes of domain objects that can be retrieved from a single table. This is exemplified in the event controller's getEventById method, where the associated Club object is initially loaded with only its ID. Full details of the Club object are loaded into memory only when its getter methods are invoked, triggering a load() method that utilizes the ClubMapper to fetch relevant details.

Lazy Initialisation



For attributes requiring multiple queries or significant computational resources, we will employ lazy initialization. This primarily applies to list objects associated with domain entities. For instance, the tickets attribute of an RSVP object is not immediately loaded into memory. Instead, it is initialized only when the getTickets() method is called, at which point the appropriate mapper retrieves and loads the list of associated tickets.

Consequences

Positive

The main benefit of a lazy loading is that it significantly reduces initial load times and memory usage by only fetching data that is needed.

The hybrid approach for lazy loading is the best option for minimizing the number of queries compared to a pure ghost or a pure lazy initialization implementation. This is because we are only conducting ghost loading on data we have already obtained from a single query, while lazy initializing other attributes that would require their own queries regardless.

Negative

Lazy loading increases the complexity of the domain model as it needs to be implemented into all domain object classes. Moreover, there is a possibility that the loaded object may contain data that are out of date with the database.

Compliance

- All domain objects with complex relationships or large data sets must implement this lazy loading strategy.
- Developers must be aware of and properly handle potential lazy loading exceptions.

Considered Alternative – Full Ghosting

Full Ghosting was our initial design choice for lazy loading. However, we opted against this approach after we decided to include attributes that contains list of domain objects for the domain object classes. This is because the load function in ghost will perform many queries to obtain the data for the list, defeating the purpose of the lazy loading pattern.

Consequences

- **Positive:** Full ghosting would be easier to implement, as all the loading logic would be in one function.
- **Negative**: Number of queries and data overhead would become excessive quickly as scale of the application grows.

Author: Kah Meng

Changelog

Version	Changes
V1.0	Initial lazy load design with ghost
V1.1	Updated design to hybrid approach to accommodate for new attributes of lists.

ADR - Identity Field Pattern

Context

The University of Melbourne Student Club Management System requires an efficient and flexible approach to managing unique identifiers for various entities in the database. We need to ensure data integrity, performance, and scalability while accommodating complex relationships between entities.

Decision

We will implement an Identity Field strategy with the following key decisions:

1. Use meaningless keys
2. Employ a combination of simple and compound keys
3. Implement table-unique keys
4. Utilize auto-generated keys

Implementation Strategy

- **Meaningless Keys:** We will implement auto-incrementing integers that are unique
- **Combination of Simple and Compound Keys:** We will implement simple keys for domain objects, and implement compound keys on association tables like admin, where it uses a composite key of ('student_id' and 'club_id').
- **Table-Unique Keys:** We will implement table unique keys by setting the tables to be auto-increment.
- **Auto-generated Keys:** We will implement auto generated keys through the setting the ids as SERIAL in PostgreSQL for the tables that uses simple keys.

Consequences

Positive

- Meaningless keys allow the system to adapt to changes in university policies or student information without compromising data integrity.
- Meaningless keys do not reveal information about the entities they represent.
- Combination of simple and compound keys accurately represents complex relationships while maintaining data integrity.
- Table-unique keys are easier to manage, especially in a system with multiple interrelated entities.
- Aligns well with the data mapper pattern, allowing independent management of entities.
- Auto-generated keys simplify the codebase and enhance maintainability.

Negative

- Potential Additional Queries: May occasionally need an extra query to retrieve a generated key.

Compliance

- All new entities must follow this Identity Field strategy.

Alternatives Considered: Meaningful Keys

Meaningful keys were considered for domain objects like email and username. We opted against it as it might lead to several issues in our application.

Consequences

- **Positive:** The keys would be easily recognizable and meaningful to users and developers. Moreover, it has potential for Natural Unique Identifiers like studentId.
- **Negative:** Changes to the meaningful data (e.g., a student's email address) would require updating the primary key, which can be complex and error-prone. It also may have potential for conflicts. Two students might share an email address, leading to uniqueness violations.

Alternatives Considered: Database-Wide Unique Keys

Database-wide unique keys were considered as an alternative to table-unique keys. This approach would ensure that each key is unique across the entire database, not just within its table.

Consequences

- **Positive:** It would ensures no duplicate keys exist anywhere in the database, which can be beneficial for data integrity.
- **Negative:** It may lead to potential Scalability Issues. As the database grows, ensuring uniqueness across all tables can become a performance bottleneck. There could be overhead in Key Generation as well. As creating keys that are unique across the entire database typically requires more complex mechanisms than table-specific keys.

Author: Kah Meng

Changelog

Version	Changes
V1.0	Decided on the implementation choices for Identity field

ADR - Foreign Key Mapping

Context

The system architecture needs to manage the relationship between Tickets and RSVP in an efficient way. The one to many relationship of RSVP to Tickets provides a challenge of how to best handle and manage the relationship not just within the domain model, but also in the database where a field in an RSVP cannot hold a collection of Ticket IDs.

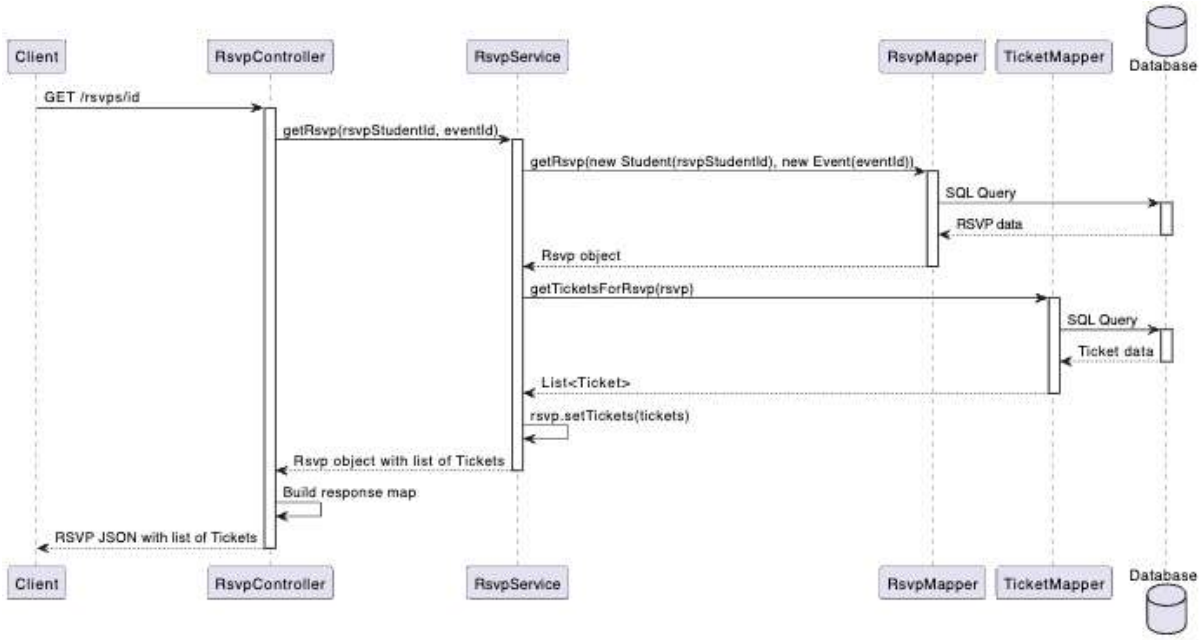
Decision

We will use the Foreign Key Mapping design pattern to map the relationship between Tickets and the corresponding RSVP.

Implementation Strategy

For the relationship where one object can refer to a collection of other objects, add foreign key values to the objects which can be in a collection. In our application, RSVPs can refer to a list of Tickets, so the composite foreign key fields rsvpStudentId and eventId fields are added to Ticket. Next, create a TicketMapper class that RSVP can call to retrieve a list of Tickets.

An example flows of retrieving Tickets using this pattern is illustrated below:



Consequences

Positive

- Avoids adding a separate join table which simplifies the database schema.
- Can query Tickets associated with a RSVP efficiently.
- Provides a clear relationship between Tickets and RSVP in the domain layer.

Negative

- Adding the foreign keys increases coupling between the domain layer and data-source layer.

- Keeping track of when a collection is updated requires another solution such as unit of work to manage corresponding changes to the database.
- Potential for inconsistent data if foreign key constraints are improperly enforced.

Compliance

- A TicketMapper class is needed to handle database operations related to Tickets.
- A method to retrieve associated tickets needs to be implemented in the RSVP class.

Alternatives Considered: Association Table

Using a separate table in the database to store and manage the relationship between Tickets and RSVPs. The added complexity to the database schema and performance impact made the team decide against using this pattern for this part of the domain.

Consequences

- Positive
  - Reduces data needed to be stored in an RSVP object.
- Negative
  - Adds an unnecessary table to the database schema.
  - More complicated queries would be needed to retrieve data related to the relationship.
  - Query performance would be impacted due to extra JOIN operations.

Alternatives Considered: Embedding RSVPs into Tickets

This alternative involves having information for the RSVP associated with a Ticket being directly in the Ticket class.

Consequences

- Positive
  - All information is in one object.
- Negative
  - Cannot manage RSVPs independently of Tickets.
  - Leads to a lot of data redundancy as RSVP information is repeated for each Ticket it is associated with which make it difficult to keep information consistent across Tickets.
  - Violates the low coupling principle of object oriented programming.

Author: Kevin Wu

Version: 1.1

Changelog:

Version	Changes
V1.0	Decision to implement Foreign Key mapping.
V1.1	Foreign Key mapping is implemented.

ADR - Association Table Mapping Pattern

Context

Our application requires an efficient way to handle many-to-many relationships between certain entities. We need a solution that can manage these complex relationships while ensuring scalability, flexibility, and performance.

Decision

We will implement the Association Table Mapping pattern to handle many-to-many relationships in our database.

Implementation Strategy

Create separate association tables in the database for many-to-many relationships. For example, admin_role table for the relationship between students and clubs they administer.

Consequences

Positive

- Allows us to link entities (e.g., students to clubs, students to events) without redundancy or data anomalies.
- Provides efficient querying and indexing of relationships, crucial for handling a large number of students, clubs, and events.
- Enables easy addition of attributes to relationships (e.g., admin role type, start date) without altering main entity tables.
- Each association has its own set of classes, promoting modularity and maintainability.

Negative:

- Increased Complexity: Requires additional tables and classes, potentially making the system more complex.
- Development Overhead: Necessitates creation and maintenance of extra classes for each association.

Compliance

- All many-to-many relationships in the system must be implemented using the Association Table Mapping pattern.
- Each association table must have corresponding controller, domain, service, and mapper classes.

Author: Kah Meng

Changelog

Version	Changes
V1.0	Initial decision on implementing Association Table Mapping pattern

ADR - Concrete Table Inheritance Pattern

Context

Our application includes entities with inheritance relationships, such as venues (online and physical) and users (students and faculty members). We need an efficient way to represent these inheritance hierarchies in our relational database while maintaining data integrity and query performance.

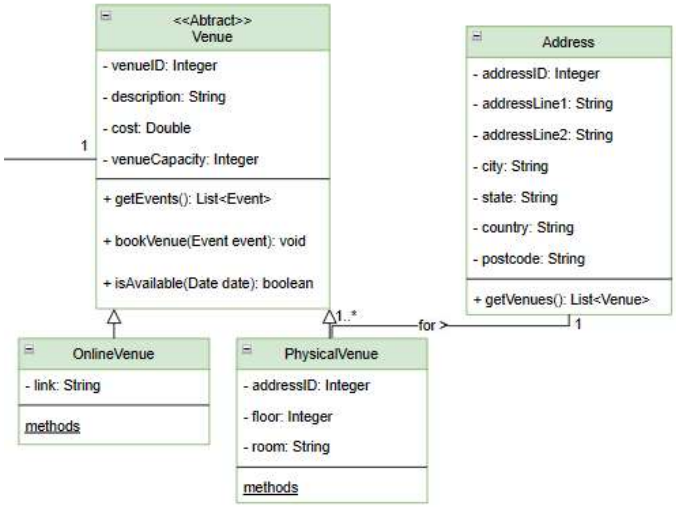
Decision

We will implement the Concrete Table Inheritance pattern to map our inheritance hierarchies to the database.

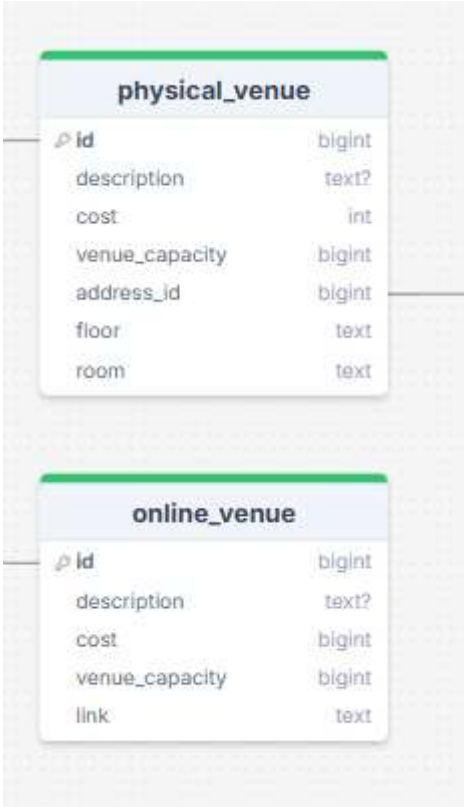
Implementation Strategy

Create separate tables for each child class in the inheritance hierarchy. The fields of the child table should include fields from both the parent class and the child class. An example of the concrete table inheritance pattern can be found in our venue entity with its OnlineVenue and PhysicalVenue child classes. The concrete tables should have its own stack of controller, service and mapper to handle their respective requests.

Domain object implementation of Venue



Database implementation of Concrete Table Inheritance



Consequences

Positive

- Concrete table inheritance is easy to implement as the mapping from domain model to database tables is straightforward.
- As all the relevant data associated with the object is on the same row, there is no need for joins to load or save a single object. Thus, improving query speed.
- Each type of entity (e.g., online venue, physical venue) can have its own specific attributes without affecting other types. This has the added benefit where all columns in all tables are relevant, avoiding null values for inapplicable attributes.

Negative:

- Common fields from the parent class are duplicated across child tables, which may lead to increased storage requirements.
- Changes to common attributes require updates to multiple tables.
- Querying across all types, like all venues, may require UNION operations or multiple queries, which can be less efficient.

Compliance

- All new entities with inheritance relationships must follow the Concrete Table Inheritance pattern.
- Controllers, Service and Mapper must be implemented for each concrete class to handle database operations.

Alternatives Considered: Single Table Inheritance

We initially considered using a single table to represent the entire inheritance hierarchy as it would be simple to implement. But ultimately decided against it later on due to the risk of data integrity and the potential issues with space complexity from the null attributes.

Consequences

- **Positive:** Simpler queries for polymorphic operations, so querying all venues will be easier
- **Negative:**
    - The biggest drawback of this method is the risk of data integrity issues that comes to nullable columns that should not be nullable (e.g., link for online venues).
    - Increased complexity in ensuring data validity in the application layer.
    - Moreover, there will be many null values for attributes not applicable to all subclasses.

Author: Kah Meng

Changelog

Version	Changes
V1.0	Initial decision on implementing Single Table Inheritance
V1.1	Changed implementation to implement Concrete Table Inheritance pattern after further discussion and initial attempts of development.

ADR - Custom Authentication and Authorisation Pattern

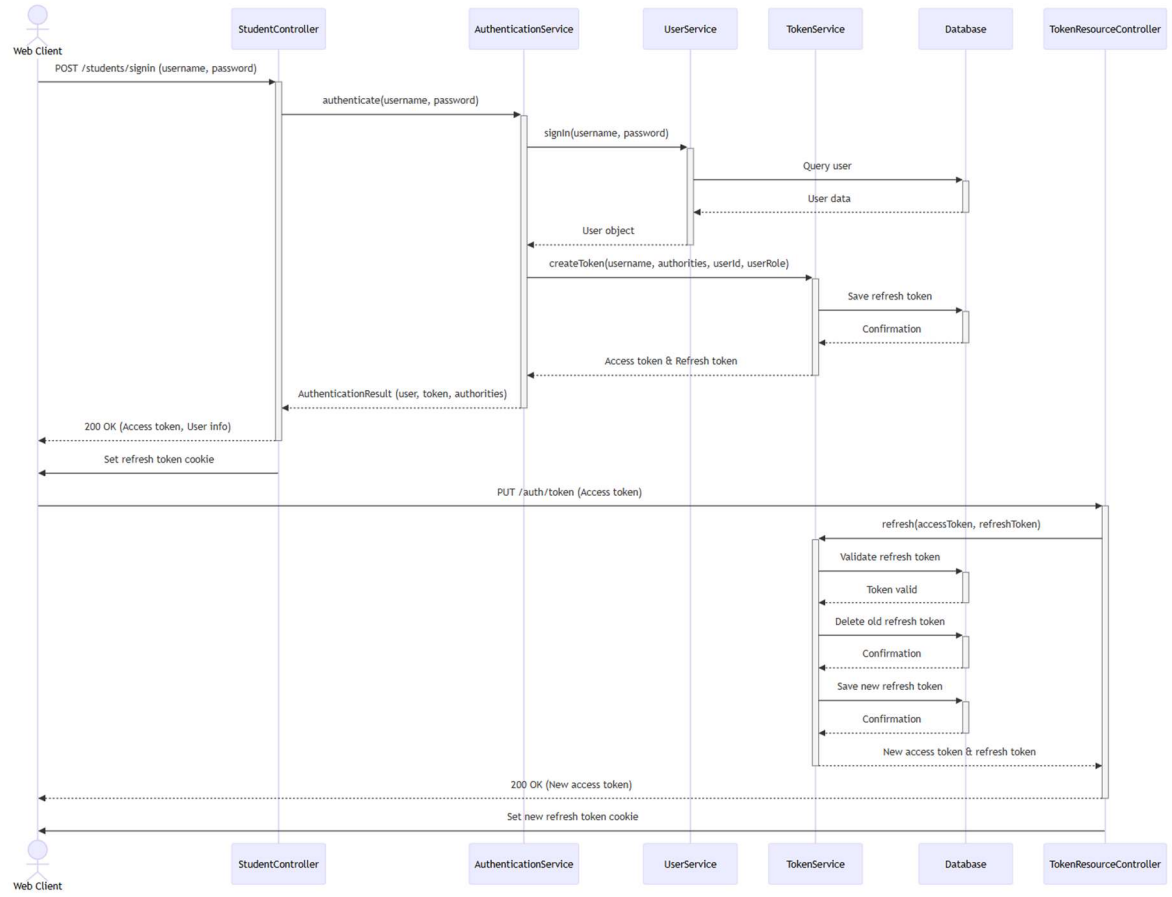
Context

The system architecture needs to manage user authentication and session handling in a secure and efficient way. The need for persistent user sessions across multiple requests presents a challenge in balancing security, performance, and user experience.

Decision

We will use JWT (JSON Web Token) based authentication with refresh tokens to manage user authentication and sessions.

Implementation Strategy



In the authentication flow, when a user logs in successfully, two tokens are issued: a short-lived JWT access token and a longer-lived refresh token. The access token is used for authenticating requests, while the refresh token is used to obtain new access tokens when they expire. The refresh tokens are stored in the database for revocation capabilities. Create a TokenService interface with a JwtTokenServiceImpl to handle token operations.

The system implements both authentication and authorization using JWT-based tokens and role-based access control.

Authentication:

1. User credentials (username and password) are sent to the StudentController's signIn endpoint.

2. The AuthenticationService validates these credentials using the UserService.

3. Upon successful validation, the TokenService generates two tokens:

    - A short-lived JWT access token containing user claims (username, userId, roles)

    - A longer-lived refresh token

4. The access token is returned to the client, while the refresh token is stored in the database and sent as an HTTP-only cookie.

5. For subsequent requests, the client includes the access token in the Authorisation header.

6. When the access token expires, the client can use the refresh token to obtain a new access token via the TokenResourceController.

The AuthenticationService coordinates this process, interfacing between controllers, UserService, and TokenService.

Authorization:

1. After authentication, the system uses role-based and permission-based authorization.

2. The JWT access token contains the user's roles and permissions.

3. The AuthenticationService provides methods like isAuthorized(), isClubAdmin(), and isEventAdmin() for controllers to check user permissions.

4. These methods use the TokenService to validate the token and extract user claims.

5. For more granular permissions, the system uses a UserPermissionChecker:

- The JwtTokenServiceImpl uses this to verify specific permissions.

- It interfaces with the StudentMapper to check database-level permissions.

6. Controllers use these authorization methods to restrict access to certain endpoints or operations.

The AuthenticationService acts as the central component, managing both authentication and authorization:

1. It handles initial sign-in and token refresh processes.

2. It provides methods for controllers to verify user authentication and check specific permissions.

3. It uses the TokenService for token operations and validation.

4. It interfaces with the UserPermissionChecker for detailed permission checks.

Controllers utilize the AuthenticationService to:

- Process user sign-in (StudentController)

- Handle token refresh (TokenResourceController)

- Verify user authentication and permissions for protected routes

Consequences

Positive

- Allows for longer user sessions through refresh tokens without compromising the security of short-lived access tokens.
- Decouples token creation and validation through the TokenService interface, allowing for easy changes in token implementation.
- Centralizes authentication and authorization logic in the AuthenticationService, promoting cleaner controller code.
- Role-based access control via JWT claims
- Refresh token rotation enhances security by limiting the lifetime of each refresh token.

Negative

- Development Time: Implementing a custom solution will require more development time compared to using existing libraries.
- Security Risks: There's a higher risk of introducing security vulnerabilities if not implemented correctly, as we don't benefit from the extensive testing of established libraries.
- Maintenance Burden: We'll need to maintain and update our authentication system, including keeping up with evolving security best practices.

Compliance

- A JwtTokenServiceImpl class is needed to handle JWT operations, implementing the TokenService interface.
- A RefreshTokenRepository is required to manage refresh tokens in the database, including methods for saving, retrieving, and deleting tokens.
- The AuthenticationService must be used for all sensitive routes to ensure proper access control, including methods for authentication, authorization, and token refresh.
- Controllers must use AuthenticationService methods (isAuthorized(), isClubAdmin(), isEventAdmin()) to verify user permissions before allowing access to protected resources.

Alternatives Considered - Third-Party Authentication Libraries

- Positive: Well-tested, regularly updated for security vulnerabilities.

- Negative: May not provide the specific functionality we need for our unique user roles and permissions.

Author: James Launder

Changelog

Version	Changes
V1.0	Initial discussion on implementation approaches with Authentication and Authorisation Pattern
V1.1	Initial decision on implementing custom Authentication and Authorisation Pattern

V1.2	Inclusion of Refresh token

ADR - State Design Pattern

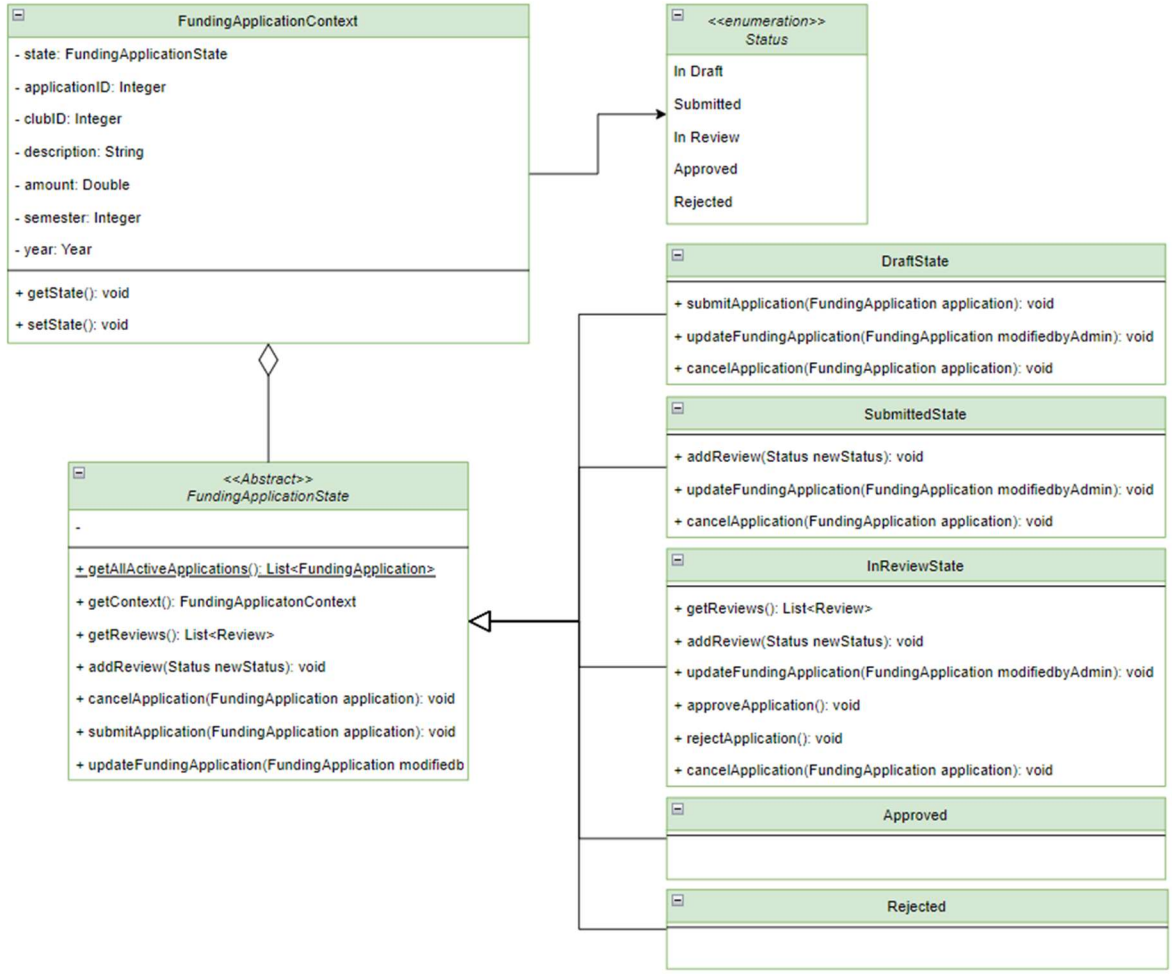
Context

For funding applications, the team noted that there would be different methods applicable that a user may execute on an application depending on what status the application is in. For example, an application can only be reviewed by a faculty member when it has a Submitted or In Review status. In our implementation, an application may be reviewed by multiple faculty members as will be a concurrency scenario for Part 3.

Decision

The State design pattern is an elegant solution for managing funding application behaviour and transitions in predefined states and will be implemented.

Implementation Strategy



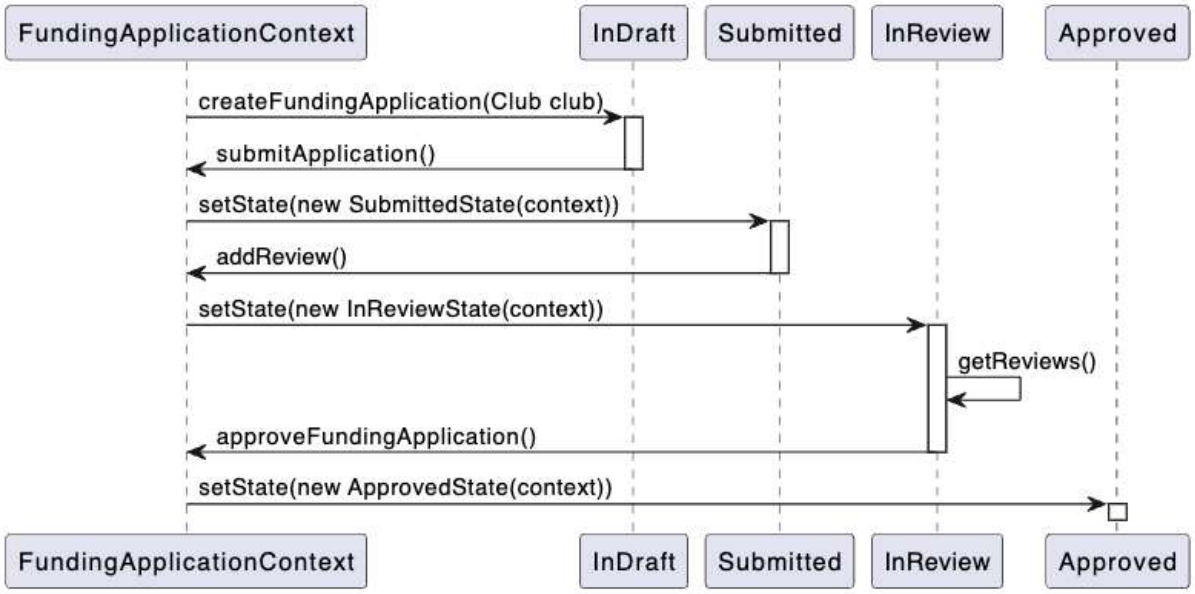
The pattern is suitable for programs where there is a finite number of states a program can be in,

and the program can only be in one state at any given time. The behaviour is also different within each unique state with transitions between states being finite and predetermined. The state pattern is similar to the idea of a finite state machine and is well suited to funding applications, with each possible status being a state an application can be in. This also allows for an elegant implementation of status specific behaviour.

Each status is represented by a concrete state class which all inherits from an abstract FundingApplicationState class. In the abstract class, each possible method is blocked. If a status allows a method, the method is overridden in the concrete class and implemented. Methods not overridden remain blocked in that state by default.

A separate FundingApplicationContext class keeps track of a specific funding application, containing a concrete state object and data related to a funding application.

The following demonstrates a flow of creating to approving a funding application, showing the transitions between different concrete status objects.



Consequences

Positive

- Encapsulates status and state specific behaviours. Making changes to existing states or adding new states can be done without changing the FundingApplicationContext class nor require changes to other existing classes.
- Helps enforce and clearly defines valid transitions between classes.
- Improves code readability by separating status behaviour and code into their respective concrete classes.
- Facilitates clearer unit testing for state specific behaviour.

Negative

- Increases the number of classes in the system as each status has its own concrete class in addition to a context and abstract class.
- Creation of multiple state objects increases memory usage.
- More complex serialisation is required to map state objects.

Compliance

- A FundingApplicationMapper is needed to handle database operations between the domain layer and the database.
- Transitions between states need to be well defined and adhered to.
- JSON serializer required to ensure the different states are represented consistently in the API queries.

Alternatives Considered: Using Enums for State Management with Conditional Logic

Consequences

We considered having if or switch statements within each possible method to allow or disallow the calling of methods depending on what status an application has. The enum will be used to keep track of the status of an application. However, whilst this approach could work, it would result in a lot of logic and nested conditional code in a single funding application class. After some research online, we found that the 'State' design pattern would be a more elegant solution for implementing the required behaviour requirements for funding applications.

- Positive
    - Less classes to manage.
    - Simpler to understand.
- Negative
    - Difficult to extend.
    - Conditional logic can become bloated and hard to understand.

Author: Kevin Wu

Version: 1.2

Changelog:

Version	Changes
V1.0	Discussion and decision to use the State design pattern.
V1.1	Implemented In Draft and Submitted states, ensuring transition logic holds.
V1.2	Defined and implemented basic valid transitions for other states.