Department of Computing and Information Systems

Software Architecture Document (SAD)

For

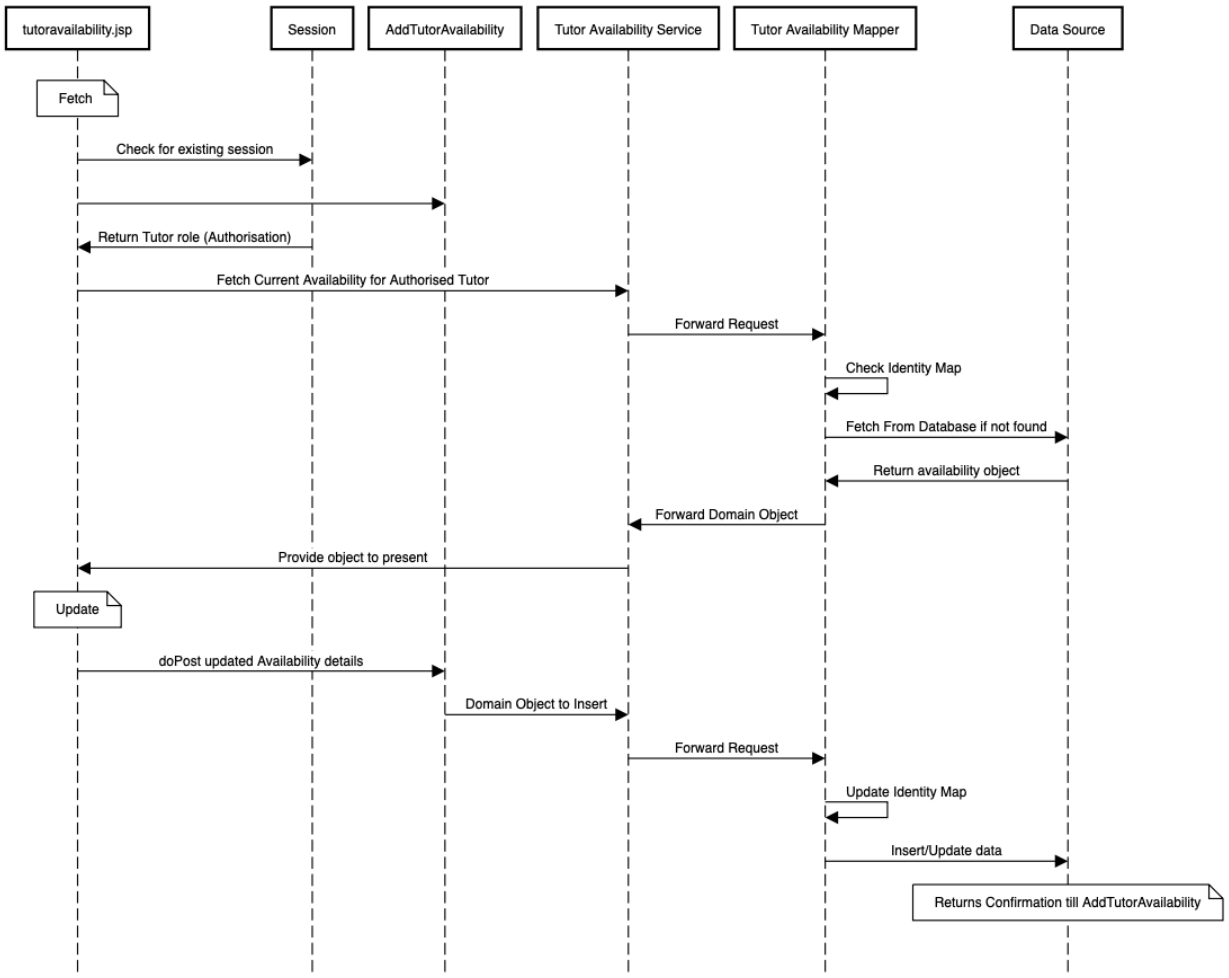
# Tutor Discovery and Booking System

**Version 4.0**
**26/10/2019**

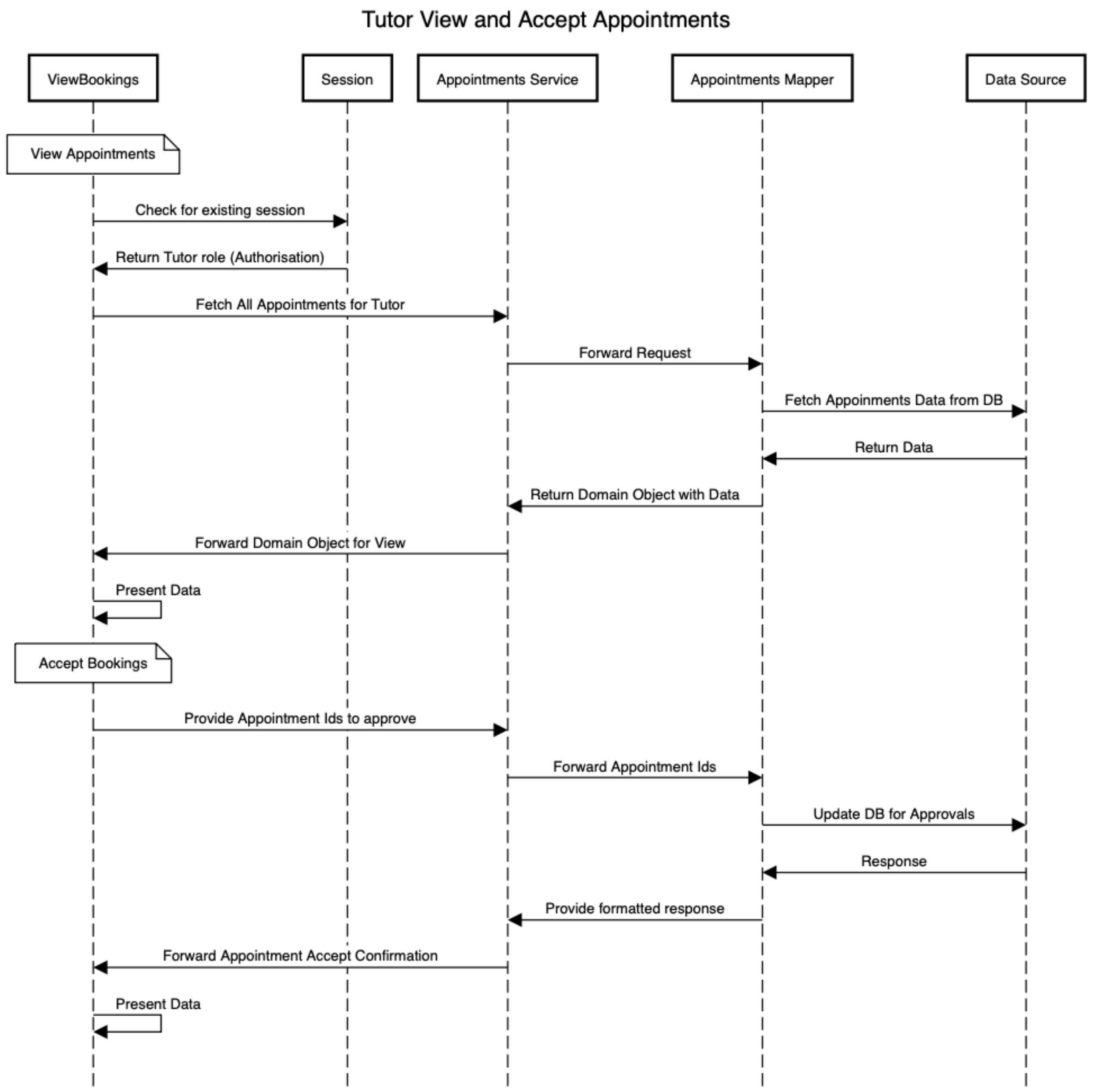
Karan Katnani: kkatnani@student.unimelb.edu.au

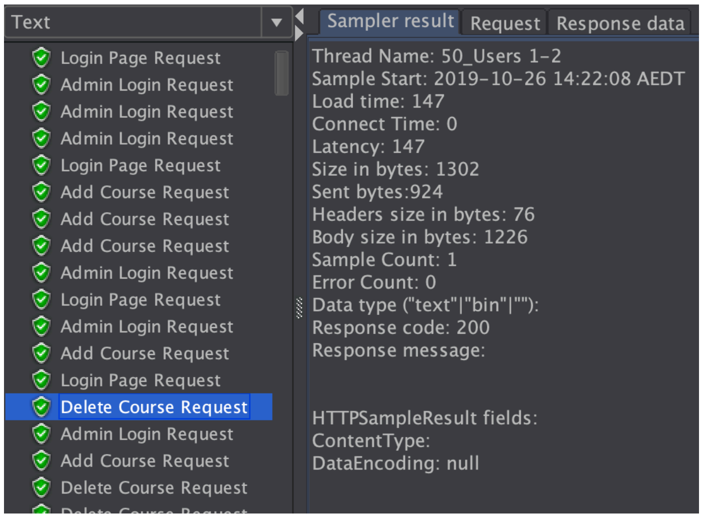
# Table of Contents

Tutor Post Availability


tutoravailability.jsp
Session
AddTutorAvailability
Tutor Availability Service
Tutor Availability Mapper
Data Source
Fetch
Check for existing session
Return Tutor role (Authorisation)
Fetch Current Availability for Authorised Tutor
Forward Request
Check Identity Map
Fetch From Database if not found
Return availability object
Forward Domain Object
Provide object to present
Update
doPost updated Availability details
Domain Object to Insert
Forward Request
Update Identity Map
Insert/Update data
Returns Confirmation till AddTutorAvailability

22


Tutor View and Accept Appointments
ViewBookings
Session
Appointments Service
Appointments Mapper
Data Source
View Appointments
Check for existing session
Return Tutor role (Authorisation)
Fetch All Appointments for Tutor
Forward Request
Fetch Appoinments Data from DB
Return Data
Return Domain Object with Data
Forward Domain Object for View
Present Data
Accept Bookings
Provide Appointment Ids to approve
Forward Appointment Ids
Update DB for Approvals
Response
Provide formatted response
Forward Appointment Accept Confirmation
Present Data



```
Login Page Request
Admin Login Request
Admin Login Request
Admin Login Request
Login Page Request
Add Course Request
Add Course Request
Add Course Request
Admin Login Request
Login Page Request
Admin Login Request
Add Course Request
Login Page Request
Delete Course Request
Admin Login Request
Add Course Request
Delete Course Request
```

```
Thread Name: 50_Users 1-2
Sample Start: 2019-10-26 14:22:08 AEDT
Load time: 147
Connect Time: 0
Latency: 147
Size in bytes: 1302
Sent bytes:924
Headers size in bytes: 76
Body size in bytes: 1226
Sample Count: 1
Error Count: 0
Data type ("text"|"bin"|""):
Response code: 200
Response message:


HTTPSampleResult fields:
ContentType:
DataEncoding: null
```

## 1. Overview

The goal of this project is to build a system that would assist students and tutors to connect with each other. The intent is to get tutors to register with their subjects and service locations, and for students to be able to find a tutor for their subject as close as possible.

### Context

As the education space gets more and more competitive, and class sizes keep expanding, the need for quality personal attention becomes more and more important. Not all students are able to cope at the same rate and require special attention outside of class to be able to catch up. At the same time, there is a plethora of individuals with some spare time, knowledge and vast experience, who would like to teach students when they can, without having to go through the process of becoming a Teacher. We decided to come up with this system to connect both students and tutors together.

### Enterprise Application

The system covers the requirements of being an enterprise application as:

- All transactions will be held in databases, which supports the property of persistent data, and can be scaled up on-demand as the volume of data increases.
- The system will be designed to support concurrent access amongst multiple data points, so that students and tutors can perform the tasks they want to, without knowing or having to worry about multiple users working in parallel. Multiple students would be able to search, view profiles and book appointments for the same tutor in parallel, and the tutor would have the option of cancelling request at their discretion.
- The system would be able to integrate with payment gateway systems to work with revenue models for the business, and also open to expansion to other student-oriented services. The architecture will ensure adequate cohesion and coupling for the components to work efficiently with one another, while promoting reusability in the case of expansion.

## 2. Features

**Features A (Student and Tutor Profiles)**

1. Students and Tutors can register themselves into the system with a profile
2. Students and Tutors can log themselves in to the system to view the dashboard
3. Students can view/update their profile, view tutor ratings/comments
4. Tutors can view/update their profile, view own tutorial session history
5. The admin can log in to the system to view the admin specific dashboard
6. The admin will need to approve new tutor profiles before they can be reflected in the system
7. The admin can block student or tutor accounts if deemed necessary

8. The admin can add courses.

**Features B (Booking System)**

1. The tutors can post their availability, by means of timeslots in calendars
2. When students select their tutor of choice, they can view the available time slots, based on the availability posted by the tutors, and current bookings.
3. The tutor will have a choice to accept or reject request for an appointment. Time slot will be temporarily blocked until appointment is confirmed. The block will be retained if appointment is confirmed or released otherwise.

## 3. Use Cases

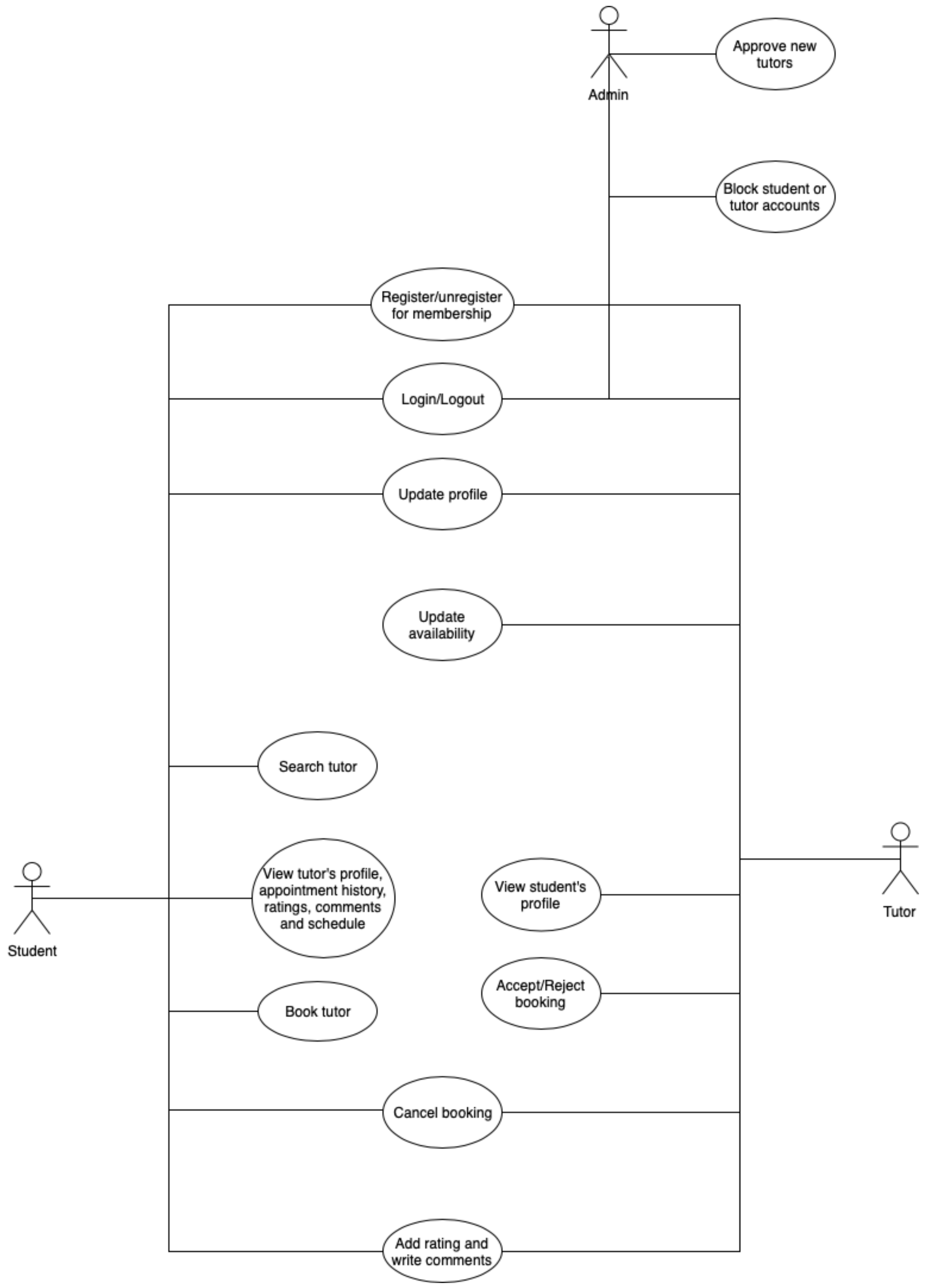

Admin
Approve new tutors
Block student or tutor accounts
Register/unregister for membership
Login/Logout
Update profile
Update availability
Search tutor
View tutor's profile, appointment history, ratings, comments and schedule
View student's profile
Accept/Reject booking
Book tutor
Cancel booking
Add rating and write comments
Student
Tutor

Figure 1: Use Cases for Tutor Management System

## 3.1. User Scenarios

1. A 10th grade student is facing difficulty with his/her Calculus class. The student wants to find a personal tutor who would be willing to come home and teach the unit. The student

finds the portal, enrols by creating an account and starts searching for a tutor teaching the subject. The student successfully finds a tutor suiting the requirements and willing to travel the distance. The student puts in a request for an available time slot, along with details of his/her home. The tutor receives an appointment request and accepts it. The tutor then goes to the student's home and completes the appointment. The tutor and student are able to provide feedback for the session.

2. With the same scenario as above, the tutor may also choose to reject the request if they choose to. The appointment will be marked as cancelled.
3. An individual with or without teaching experience would like to share their knowledge by assisting students, in exchange for money. The individual has completed a Master's Degree in Computer Science and has worked as a Full stack developer with experience in various languages, and computing disciplines. The individual creates an account, where she lists all the capabilities and distance that she is willing to travel. A student requiring similar expertise creates an appointment request at an available time slot, which the tutor approves and then fulfills the appointment, getting paid the hourly rate posted on the profile.

# 4. Feature A & B

## 4.1 Domain Class Diagram

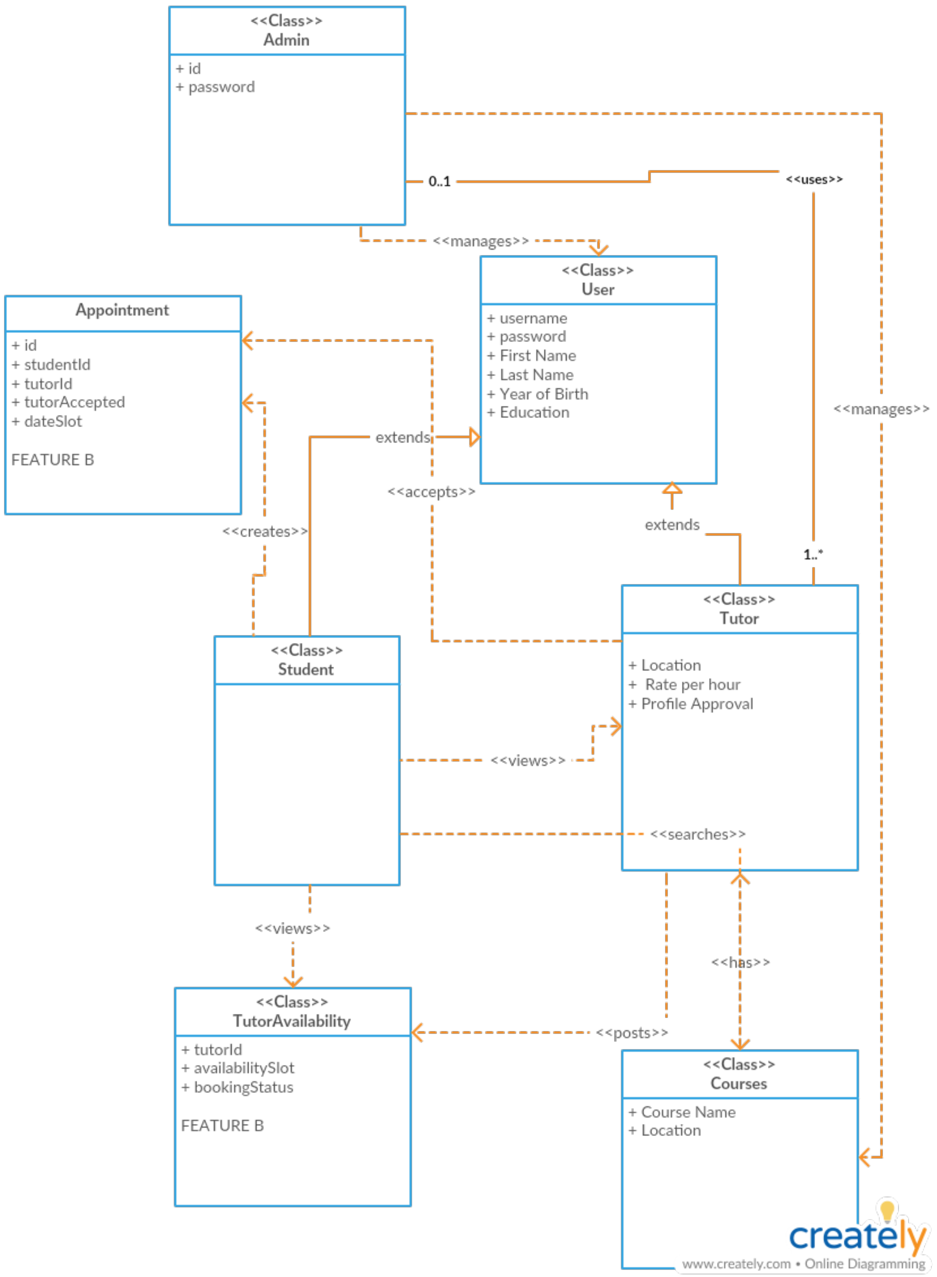


Figure 2: Domain Class Diagram for Feature A & B

Figure 2 above shows the domain level architecture of the system, designed for Features A & B.

## 4.2. Architecture

Taking the requirements of an enterprise system, along with the features and use cases above, we have implemented what is essentially the extended 3 tier layer, as described in the diagram below:

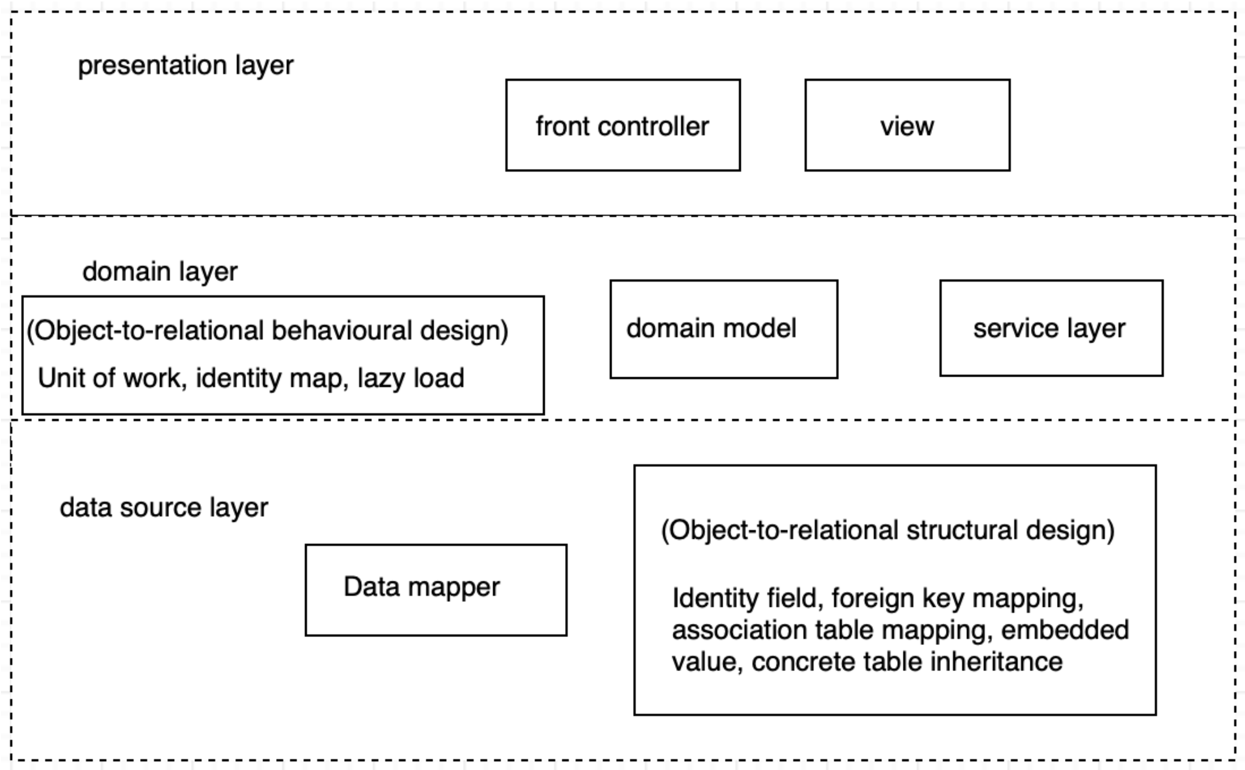

presentation layer
front controller
view
domain layer
(Object-to-relational behavioural design)
Unit of work, identity map, lazy load
domain model
service layer
data source layer
Data mapper
(Object-to-relational structural design)
Identity field, foreign key mapping, association table mapping, embedded value, concrete table inheritance

Figure 2: System Architecture

The roles and responsibilities of each component in the architecture have been described below:

- Presentation Layer: This layer is where all the user interaction takes place. Templates and views are created for the user to view information and input any data. All user requests are routed onwards from here, to the rest of the system
- Domain Layer: The domain layer is a collection of entity objects and related business logic that is designed to represent the **enterprise business model**. It consists of 2 parts:
    - Domain Model: This component deals with all the entities that exist within the system. Simply, it deals with business objects, their attributes and their relationships

    - Service Layer: This component contains the business logic of the system, implementing the tasks to be performed on the domain objects. It does not directly deal with the data source (database) though
    - Behavioural Design Patterns: The patterns like Unit of Work, Lazy Loading and Identity Map have been implemented to optimize and regulate unnecessary transactions with the data source layer.

- Data Source Layer: This layer deals with handling the data moving in and out of the persistent data store (database). The service layer issues requests for data as and when required, which is handled by this layer. It consists of:
    - Data Mapper: This component is essentially the gateway, providing a mapping between the database and domain objects.
    - Database: This is where all the persistent data of the enterprise system is held.
    - Structural Design Patterns: The design patterns listed have been implemented on the database to cleanly optimize storage and retrieval for the business system.

## 4.3. Class Diagram

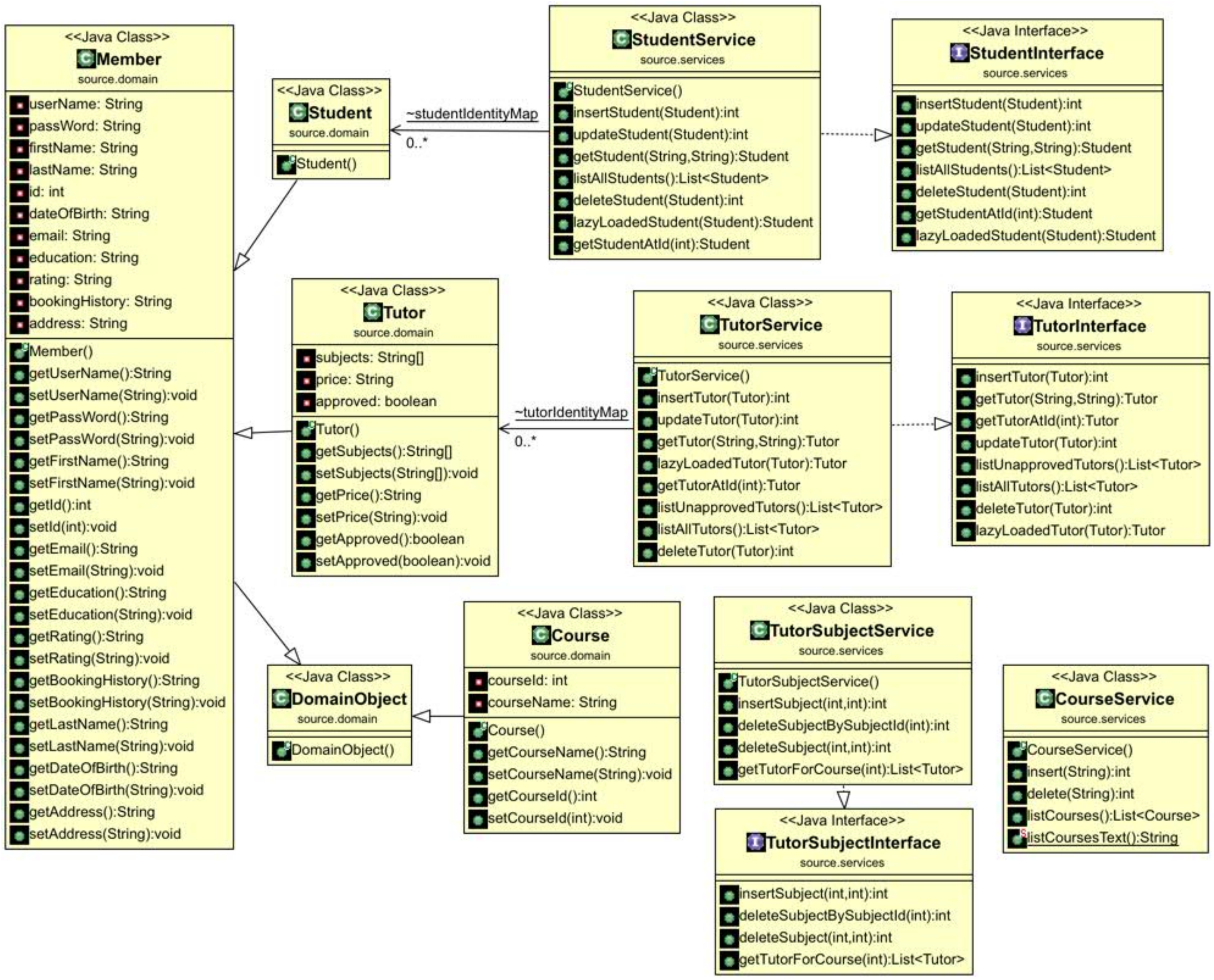


Figure 3: 1st half of Class Diagram

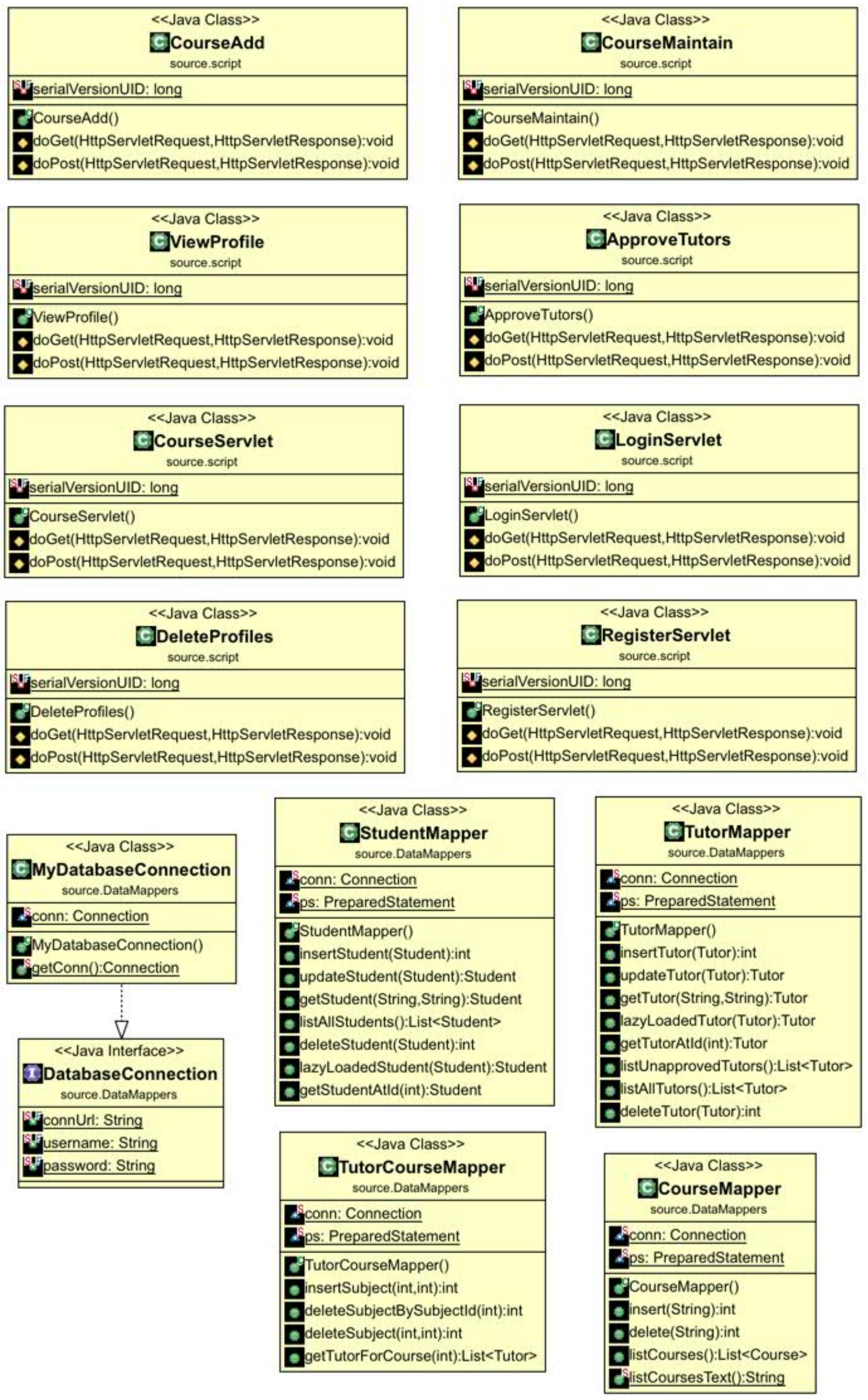


Figure 4: 2nd half of Class Diagram

## 4.4 Component Diagram

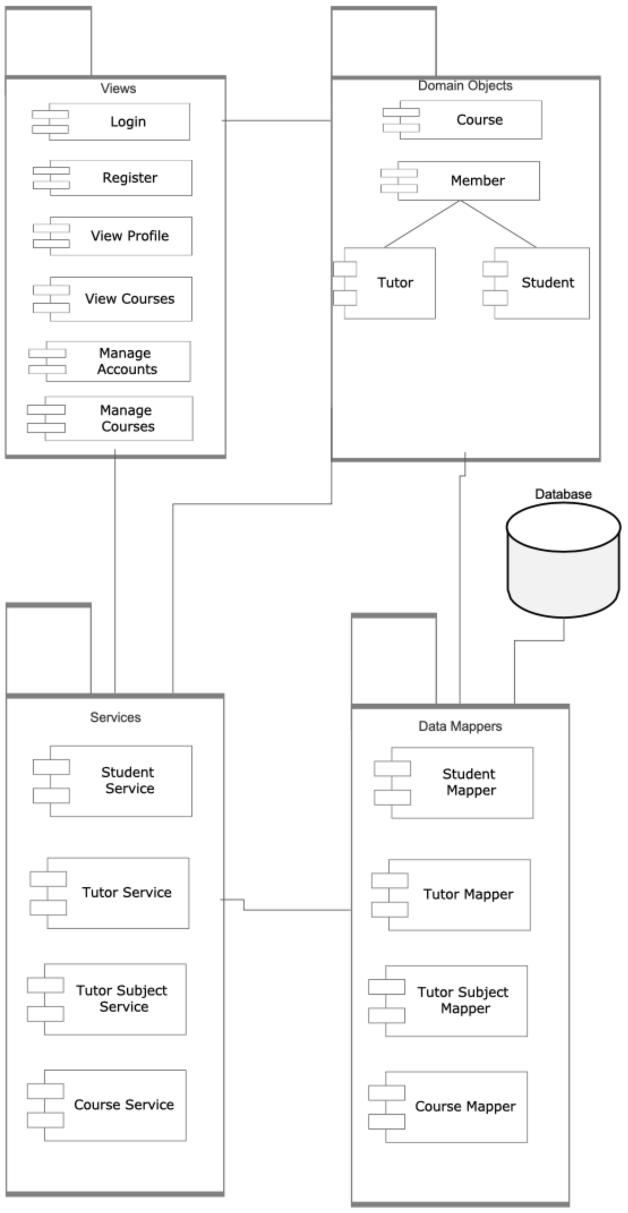


Figure 5: Component Diagram

The diagram above shows a high level view of all the components in the system. The components within each layer in the architecture have been shown here.

## 4.5 Interaction Diagram

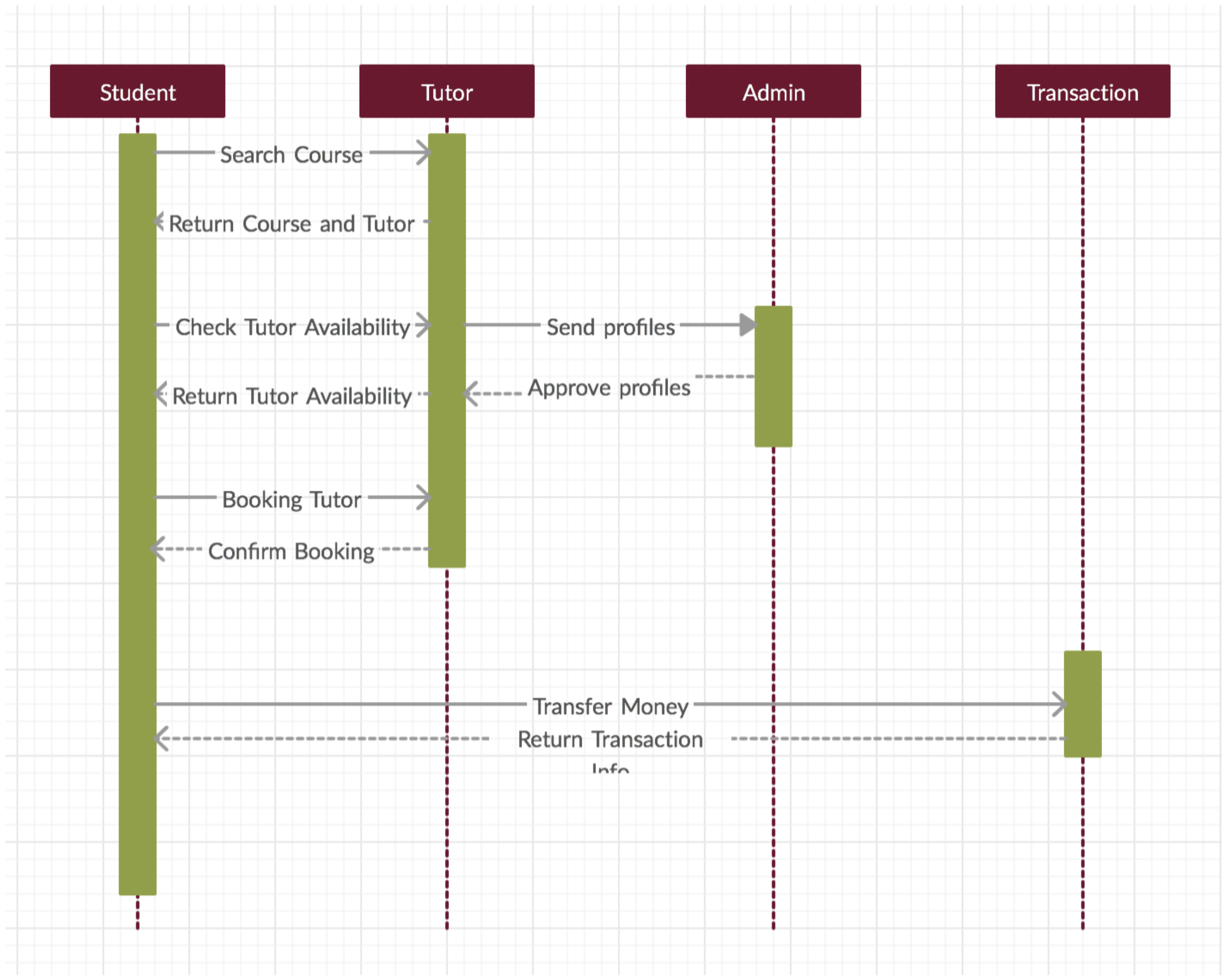


Figure 6: Interaction Diagram of the system

## 4.6 Design Patterns Implementation

In Feature A, the following patterns have been implemented:

• Domain Model and Service Layer

To make the system understandable and maintainable, we organised the domain logic with domain model.

Domain model pattern is an ideal candidate for object-orientation, as each object takes the part of the logic that is related to it. (Our class diagram represents the domain model.)

In comparison, transaction script patterns follows a procedural style of development rather than object oriented programming approach, thus it may have lower level of code duplication. With table module, we can gain much more structure because the domain logic is organized around tables rather than procedures. However, it is not as scalable and extensible as domain model pattern. Table module pattern is appropriate for applications which make heavy use of tabular data though.

With service layer, applications can be more maintainable as services that contain a mixture of different types of logic can be divided into layers where each layer contains the same type of logic. Service layer makes it easier to manage the services. In our design, all of the business logic is held in the services, in a fairly abstracted manner. It takes the request from the user, handles it if possible, or requests the data mapper layer otherwise, for data from the database.

• Data Mapper

To decouple data objects from database access layer, we use data mapper to query of the data-source layer from the domain layer.

*Data mapper* keeps objects and the database independent of each other and the mapper itself. For example, in our design, the DataMappers package contains all the data mappers, which query the database to load data objects. The services in the domain layer call these mappers to pull data when required.

Compared to *table data gateway*, *data mapper* is more compatible with domain model. (With table data gateway, we need to create one object for each table in the database.) There are also two other patterns in data-source layer: row data gateway and active record. Row data gateway is similar to table data gateway (the difference is that it creates one object per row rather than per table), however, it does not contain domain logic methods. Active record pattern is an extension of the row data gateway pattern which contains domain logic in its objects, however, it is also not appropriate for our application because of its domain logic coupling, database coupling, and lack of scalability.

• Unit of Work, Identity Map, Lazy Load

In order to efficiently get objects to load from, and save to the database, we implemented unit of work, identify map, and lazy load. Each of the three patterns solves a different problem to the other, thus they can be used together.

*Unit of work* is an object that aims to reduce the amount of data that is written back to the database by only writing back what has changed. It keeps track of everything we do that can affect the database, such as new object created, existing object updated or deleted. Unit of work maintains a list of objects and sends the updates as one transaction to the database in order to reduce small database calls. In our design, the unit of work pattern is used to handle multiple consecutive commits that go through. One of the admin functions is to delete courses, which is where multiple queries may be required to flush each course from the course database. They have been clubbed together as a single unit of work, so that only if all the courses could be deleted, the deletes get committed.

*Identity map* keeps a record of every loaded object in a map (one map per database table) to ensures that each object gets loaded only once. Whenever we want to load an object from the database, we check the identity map first to see if we already have it. In our design, Identity Maps have been used to hold objects for students and tutors domain classes, that have already been loaded before. The services for tutors and students first check the Identity Maps, and call the Data Mapper layer only if the object is not found in the identity map.

*Lazy load* is the inverse of unit work, which aims to reduce the amount of data read from database by only reading what it needs. With lazy loading, we delay the loading of an object until the point where we need it. For example, the initialization of an object occurs only when it is actually needed. In our design, student and tutor objects get partially loaded when logging in, to verify username and password. This object gets saved in the Identity Map which is then queried for other functions. If the current state of the object in the Identity Map does not contain the required attributes, the DataMapper is called to load the remaining attributes in to the object, and the Identity Map for later use.

• Identity field, foreign key mapping, association table mapping, embedded value, and class table inheritance

In order to deal with the mapping problem between domain model and its relational database, we implemented five object-to-relational structural design patterns: identity field, foreign key mapping, association table mapping, embedded value, and concrete table inheritance.

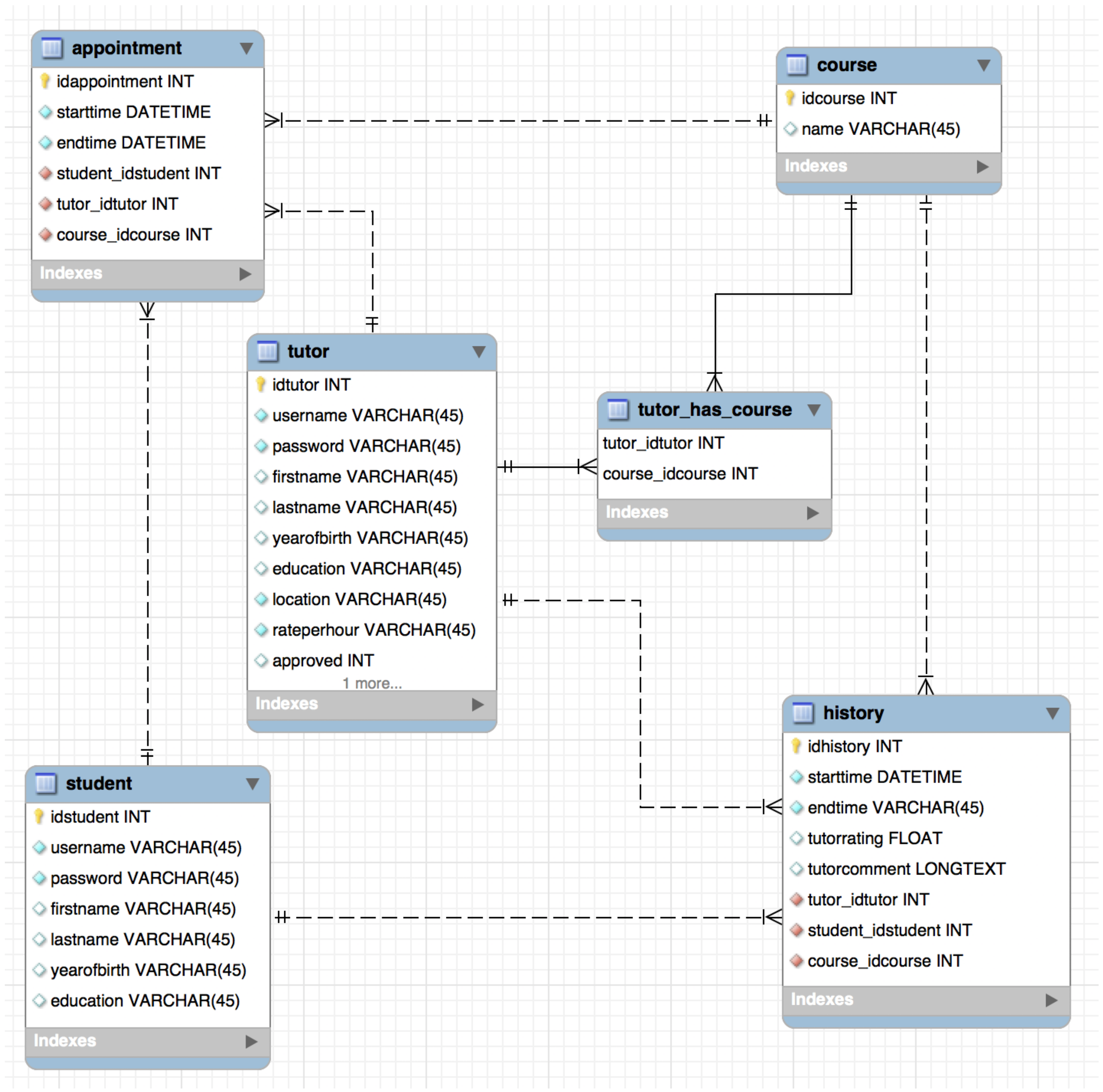


Figure 7 : Database Design ERD

Identity field: in our design, every object has a database ID field (which can increment automatically) to map an in-memory object to a database row.

Foreign key mapping: we implemented foreign key mapping in *appointment* table and *history* table. It maps the object references (student, tutor, course) to foreign keys (idstudent, idtutor, idcourse) respectively.

Association table mapping: while foreign key mapping is used for one-to-many associations, association table mapping can be used for many-to-many associations. We created an extra table *tutor_has_course* which records the association between tutor and course with foreign keys and use association table mapping to map the multivalued field to this table.

Embedded value: in our design, we have an *appointment* object with links to a time range object. As time range does not make sense as tables in a database, we map it to fields in the *appointment* table rather than makes a new table for it.

Concrete table inheritance represents an inheritance hierarchy of classes with one table per concrete class. Compared to single table inheritance which represents an inheritance hierarchy of classes as a single table, concrete table inheritance has better table design and can spread load to different tables. Compared to class table inheritance which represents an inheritance hierarchy of classes with one table for each class, concrete table inheritance has no join, so global finding can be a problem.

## 5.0 Components for Feature B

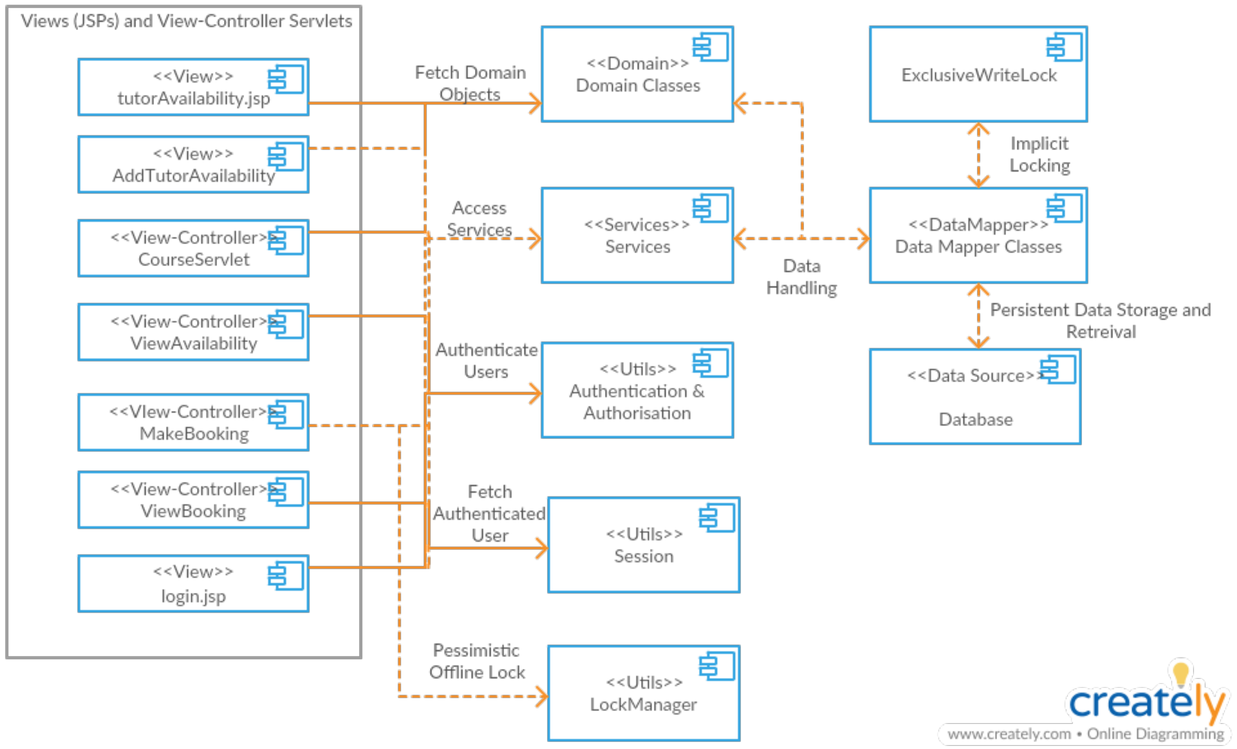

Views (JSPs) and View-Controller Servlets
<<View>> tutorAvailability.jsp
<<View>> AddTutorAvailability
<<View-Controller>> CourseServlet
<<View-Controller>> ViewAvailability
<<VIew-Controller>> MakeBooking
<<View-Controller>> ViewBooking
<<View>> login.jsp
Fetch Domain Objects
Access Services
Authenticate Users
Fetch Authenticated User
Pessimistic Offline Lock
<<Domain>> Domain Classes
<<Services>> Services
<<Utils>> Authentication & Authorisation
<<Utils>> Session
<<Utils>> LockManager
Data Handling
ExclusiveWriteLock
Implicit Locking
<<DataMapper>> Data Mapper Classes
Persistent Data Storage and Retreival
<<Data Source>> Database
creately
www.creately.com • Online Diagramming

Figure 8: Component diagram for feature B

## 5.1 Interactions for Feature B

Interactions with the Domain Objects have not been shown, to reduce the complexity of the interaction diagrams.

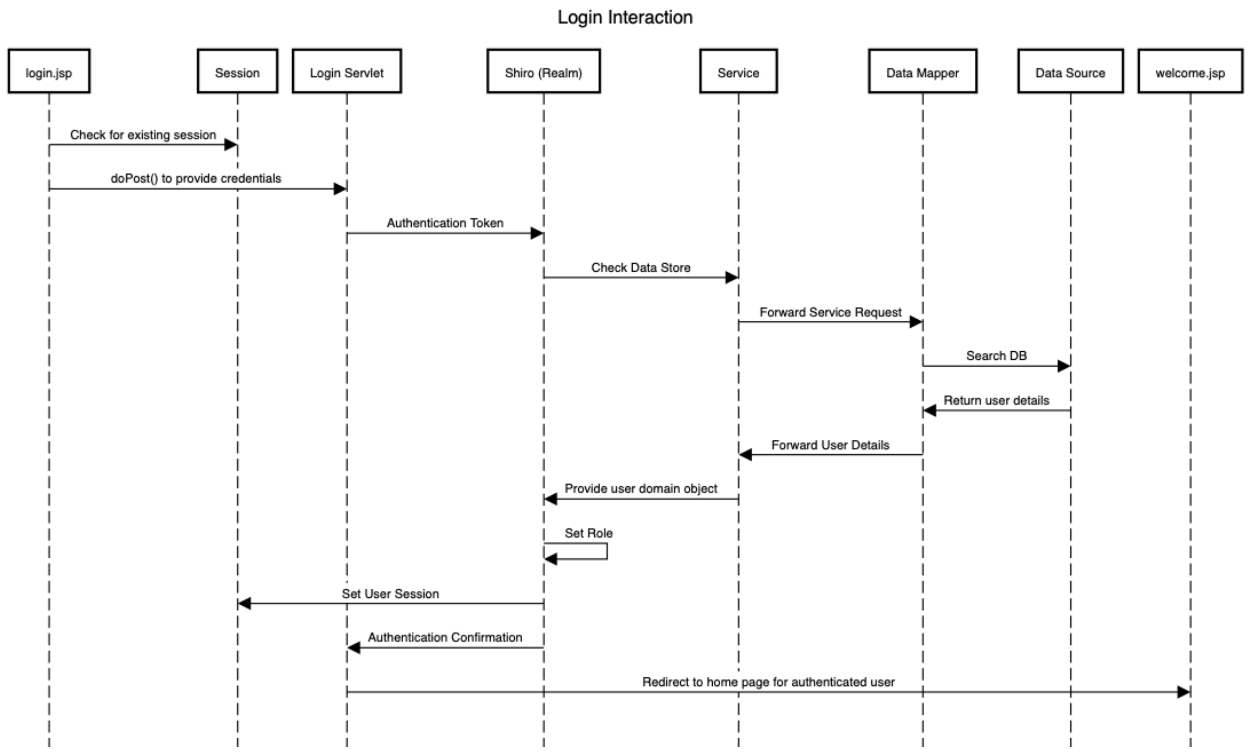


Figure 9: Interactions to establish Authentication and Session for User

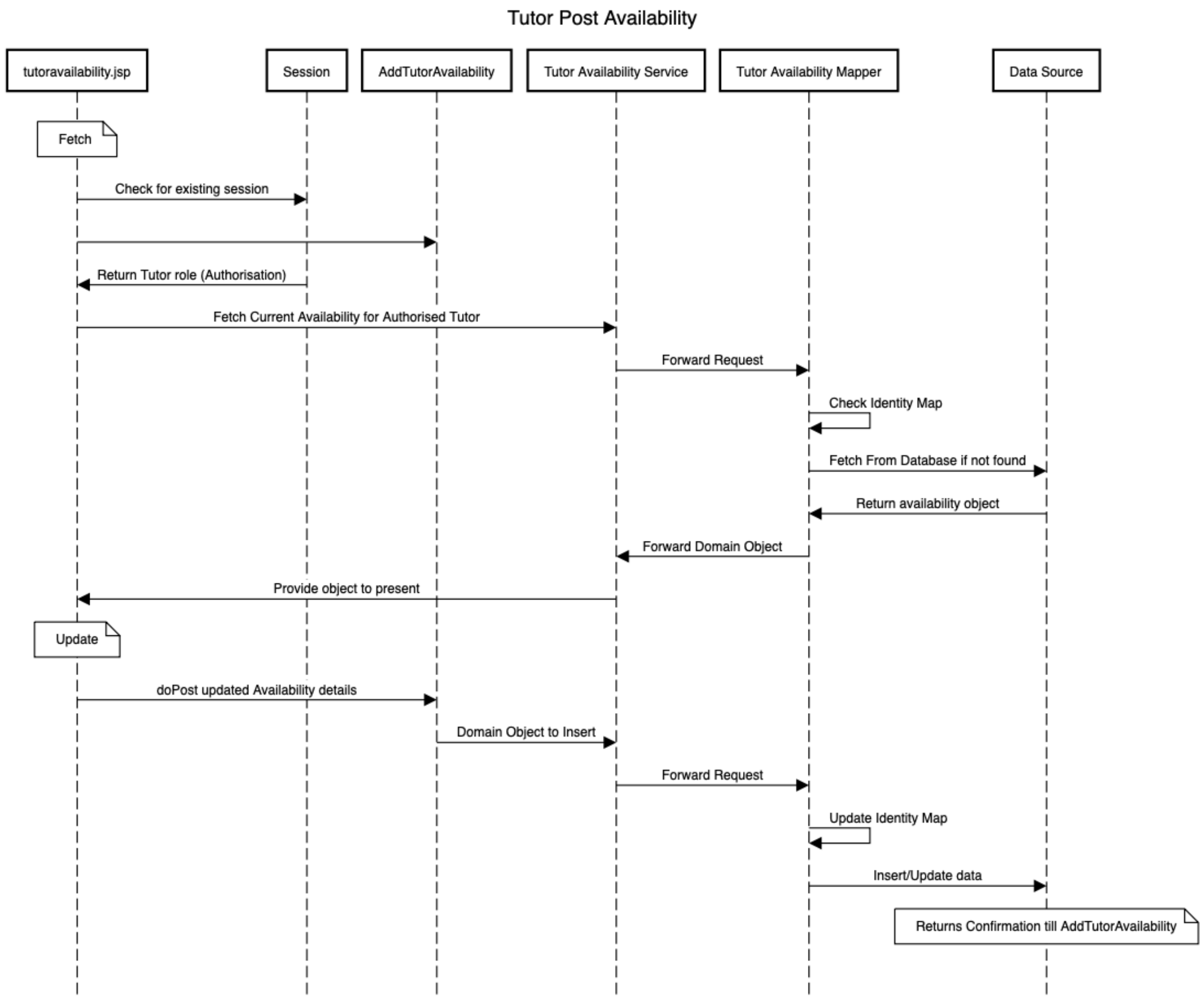

Tutor Post Availability
tutoravailability.jsp
Session
AddTutorAvailability
Tutor Availability Service
Tutor Availability Mapper
Data Source
Fetch
Check for existing session
Return Tutor role (Authorisation)
Fetch Current Availability for Authorised Tutor
Forward Request
Check Identity Map
Fetch From Database if not found
Return availability object
Forward Domain Object
Provide object to present
Update
doPost updated Availability details
Domain Object to Insert
Forward Request
Update Identity Map
Insert/Update data
Returns Confirmation till AddTutorAvailability

Figure 10: Interaction for Tutor viewing availability (if exists) and updating them

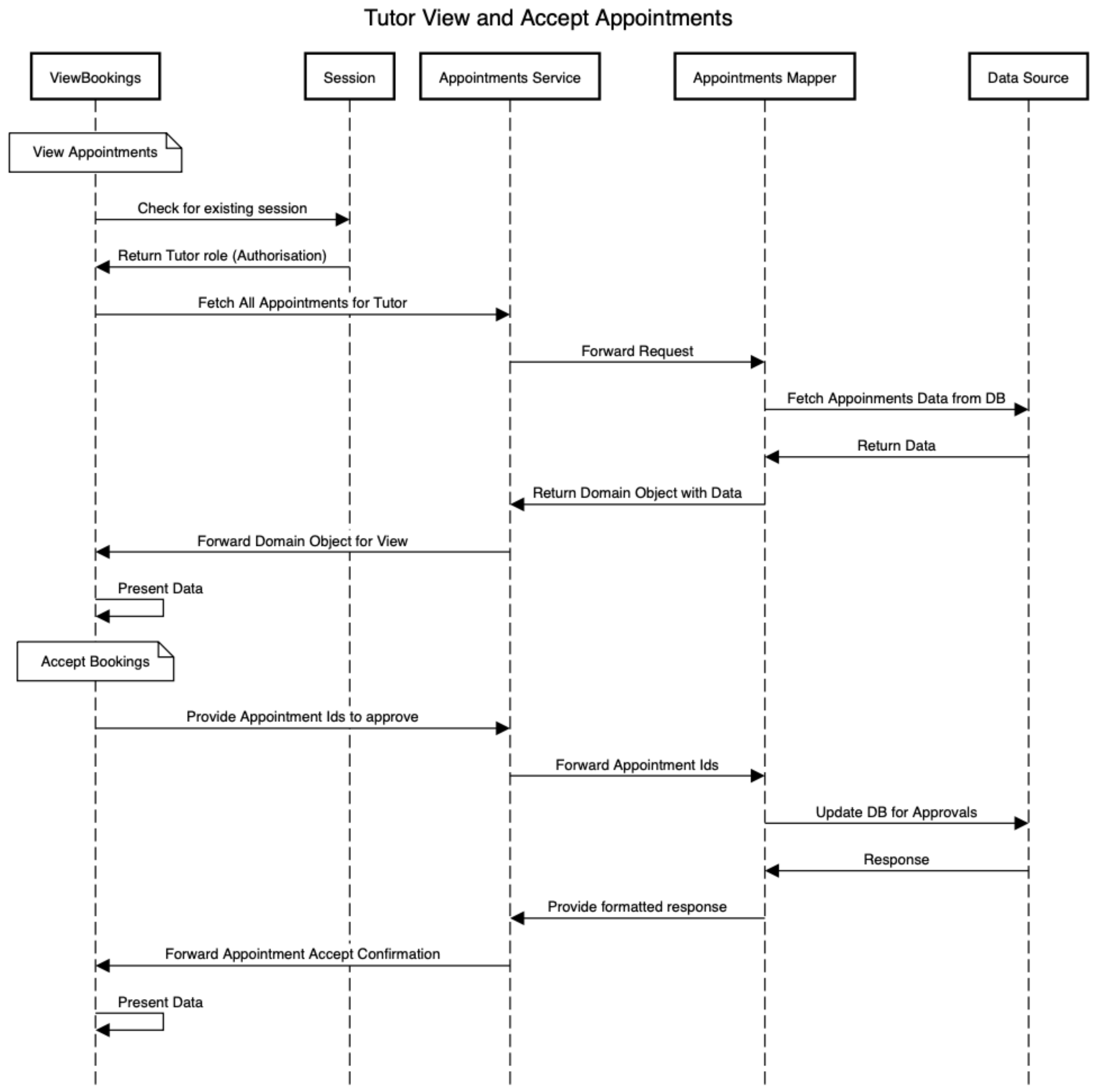

Tutor View and Accept Appointments
ViewBookings
Session
Appointments Service
Appointments Mapper
Data Source
View Appointments
Check for existing session
Return Tutor role (Authorisation)
Fetch All Appointments for Tutor
Forward Request
Fetch Appoinments Data from DB
Return Data
Return Domain Object with Data
Forward Domain Object for View
Present Data
Accept Bookings
Provide Appointment Ids to approve
Forward Appointment Ids
Update DB for Approvals
Response
Provide formatted response
Forward Appointment Accept Confirmation
Present Data

Figure 11: Tutor can view and accept booking requests by students

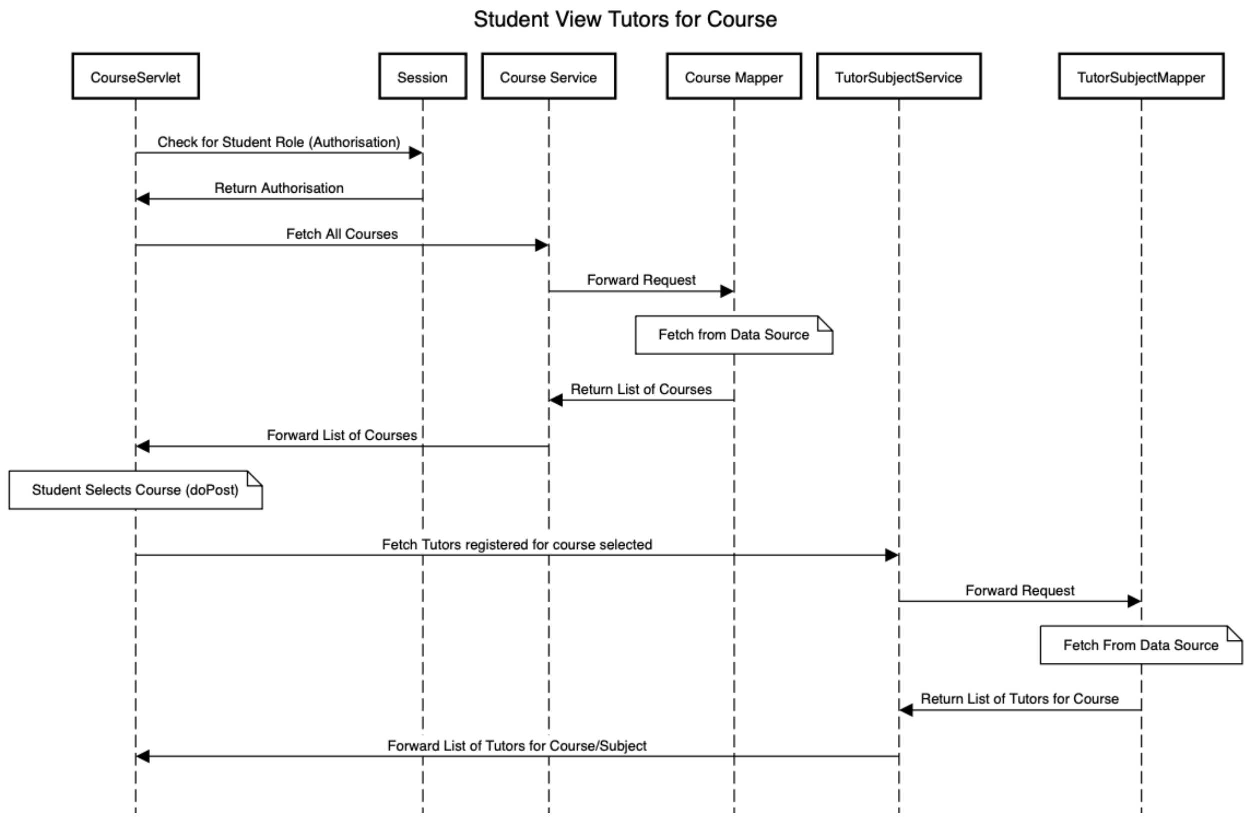


Figure 12: Students view tutors for courses they want to study

The Figure above is part of feature A, but has been included to provide context and continuity.

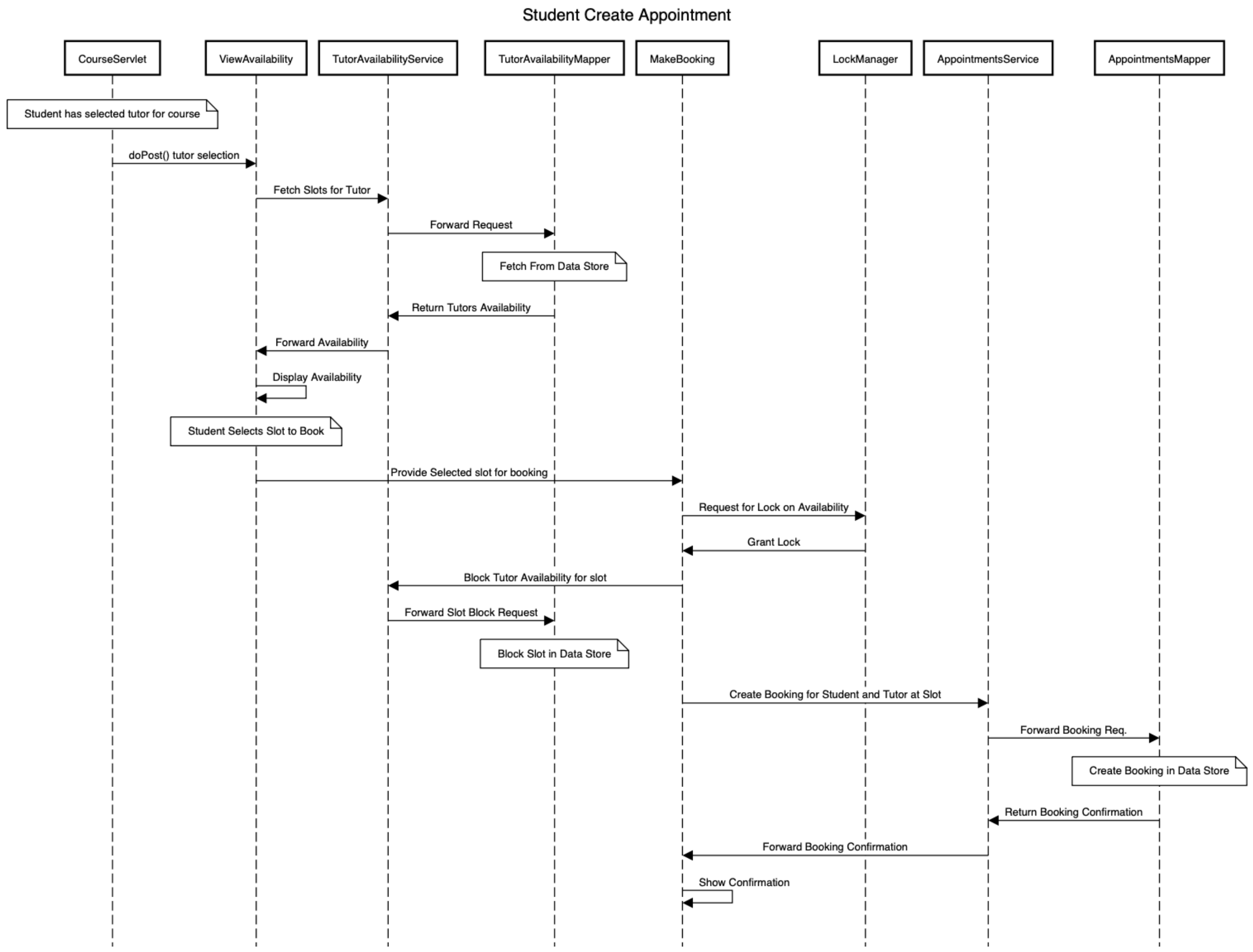

Student Create Appointment
CourseServlet
ViewAvailability
TutorAvailabilityService
TutorAvailabilityMapper
MakeBooking
LockManager
AppointmentsService
AppointmentsMapper
Student has selected tutor for course
doPost() tutor selection
Fetch Slots for Tutor
Forward Request
Fetch From Data Store
Return Tutors Availability
Forward Availability
Display Availability
Student Selects Slot to Book
Provide Selected slot for booking
Request for Lock on Availability
Grant Lock
Block Tutor Availability for slot
Forward Slot Block Request
Block Slot in Data Store
Create Booking for Student and Tutor at Slot
Forward Booking Req.
Create Booking in Data Store
Return Booking Confirmation
Forward Booking Confirmation
Show Confirmation

Figure 13: Student creates booking request for tutor

## 5.2 Design Patterns Implementation Part 2

Along with Feature B, the following architectural and design patterns have been implemented:

- Presentation Layer:

A hybrid approach of MVC (Model-View-Controller) and MVVC (Model-View-View Controller) has been used in the context of this project to deal with the user views. The reason for the hybrid approach is that there were multiple views which required scripting, and the usage of scriptlets in view files (jsp) need to be avoided. In pages where scriptlets were getting heavier, they were changed to view controllers (servlets serving html) served using 'GET' requests.

- Security:

Authentication and Authorization Enforcers:
Apache's Shiro library has been used to enforce Authentication and Authorization across the platform. The login controller passes a login token to Shiro, which is connected via Data Mappers to the User Store at the Data Source layer. If the user details exist, the user's domain object and role is fetched, and returned to the login servlet.

Validation:
No explicit data validation has been applied as part of this architecture. The risk of malicious data entry (SQL Injection) has been handled at the data mapper layer, where all inputs have been entered into the sql statements using the JDBC's Prepared Statements. Available resources suggest that constructing sql statements using string concatenation leave the system exposed to SQL injection, and using Prepared Statements internally handles it.

Integrity and Confidentiality:
Networks are known to be inherently insecure, and data moving across networks is at high risk of Man in the middle attacks. Transport Layer Security (TLS) has been implemented to ensure that the data moving from the browser to server and vice-versa moves in an encrypted format, to protect against any such vulnerabilities during data transport. A self-signed SSL certificate has been set within the Apache Tomcat Server's KeyStore, for the client browser to be able to establish a secure session with the server.

- Sessions:

In order to allow an authenticated user to access parts of the system that it is authorized to do, session data needs to be maintained across the lifetime of the logged in user. Once the user is authenticated by Shiro, the user's domain object, along with the respective role is retained in a session class. This session class is accessed by all the scripts to check for role authorities, and authentication, before allowing the user to access functionalities.

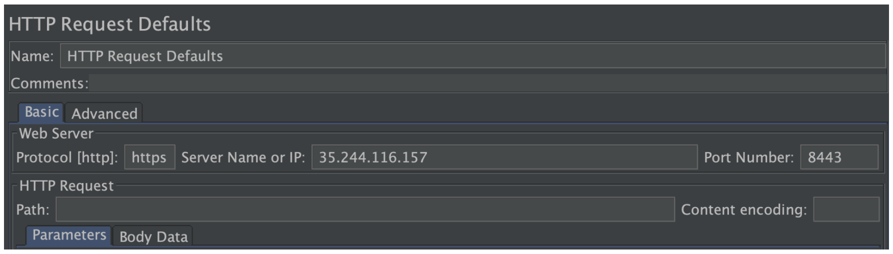
- Concurrency:

Implicit Locks in the data mapper layer have been utilized for Appointment creation and deletion, where the table gets locked when bulk operations are being run on the appointments table. Such locking has been used to avoid the need for explicit locking at the script level, to provide more flexibility to the programmers
A pessimistic offline lock has been implemented as an explicit lock, on the tutor availability, to create a booking for a student with a tutor. This has been done to avoid concurrency problems where 2 students might try to book the same tutor for the same slot, at the same time. The pessimistic lock was used to prevent conflicts before they can even take place, as opposed to the optimistic lock, which will handle the conflict only if it actually happens.

## 6.0 Testing

### Test Plan

The system has been performance tested in Junit, where load and response times from the deployment server have been tested for [Figure 14].

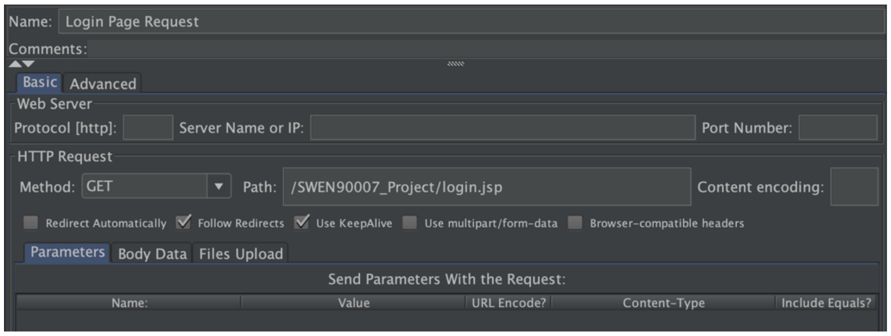


HTTP Request Defaults
Name: HTTP Request Defaults
Comments:
Basic | Advanced
Web Server
Protocol [http]: https  Server Name or IP: 35.244.116.157  Port Number: 8443
HTTP Request
Path:  Content encoding:
Parameters | Body Data

Figure 14: Server Configuration in JMeter

The admin role was selected for testing, for which the following operations were tested:

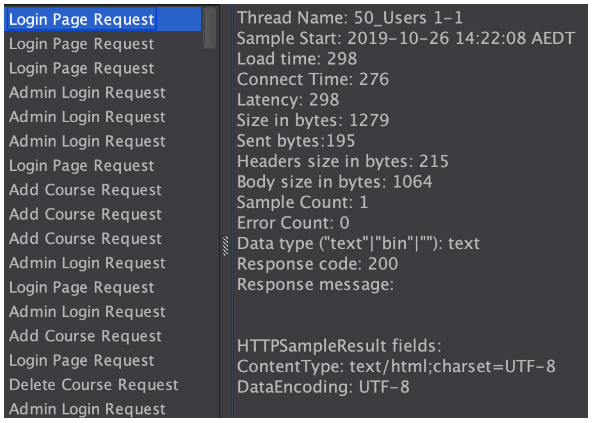
- The login view is fetched [Figure 15 & 16]



Name: Login Page Request
Comments:
Basic | Advanced
Web Server
Protocol [http]:  Server Name or IP:  Port Number:
HTTP Request
Method: GET  Path: /SWEN90007_Project/login.jsp  Content encoding:
Redirect Automatically  ✓ Follow Redirects  ✓ Use KeepAlive  Use multipart/form-data  Browser-compatible headers
Parameters | Body Data | Files Upload
Send Parameters With the Request:

| Name: | Value | URL Encode? | Content-Type | Include Equals? |
|---|---|---|---|---|

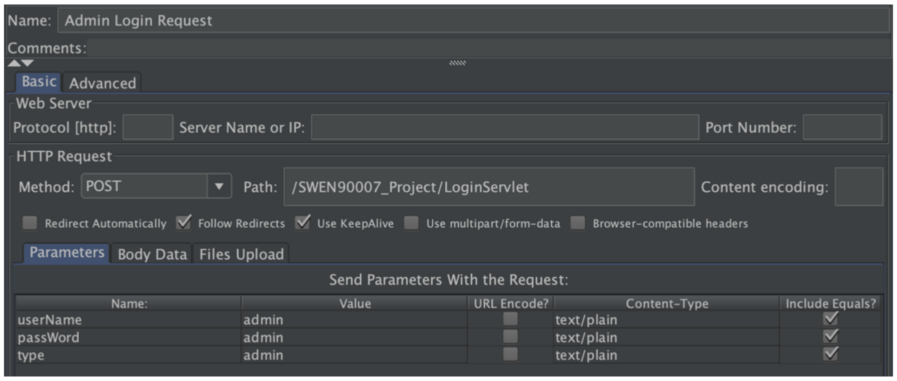
Figure 15: HTTP Request to fetch login page



```
Login Page Request       Thread Name: 50_Users 1-1
Login Page Request       Sample Start: 2019-10-26 14:22:08 AEDT
Login Page Request       Load time: 298
Admin Login Request      Connect Time: 276
Admin Login Request      Latency: 298
Admin Login Request      Size in bytes: 1279
Login Page Request       Sent bytes:195
Add Course Request       Headers size in bytes: 215
Add Course Request       Body size in bytes: 1064
Add Course Request       Sample Count: 1
Admin Login Request      Error Count: 0
Login Page Request       Data type ("text"|"bin"|""): text
Admin Login Request      Response code: 200
Add Course Request       Response message:
Login Page Request
Delete Course Request    HTTPSampleResult fields:
Admin Login Request      ContentType: text/html;charset=UTF-8
                         DataEncoding: UTF-8
```

Figure 16: Response for 1 Thread fetching login page

- The admin user is authenticated, and session and role are established. [Figure 16]



Name: Admin Login Request

Comments:

Basic | Advanced

Web Server

Protocol [http]: Server Name or IP: Port Number:

HTTP Request

Method: POST Path: /SWEN90007_Project/LoginServlet Content encoding:

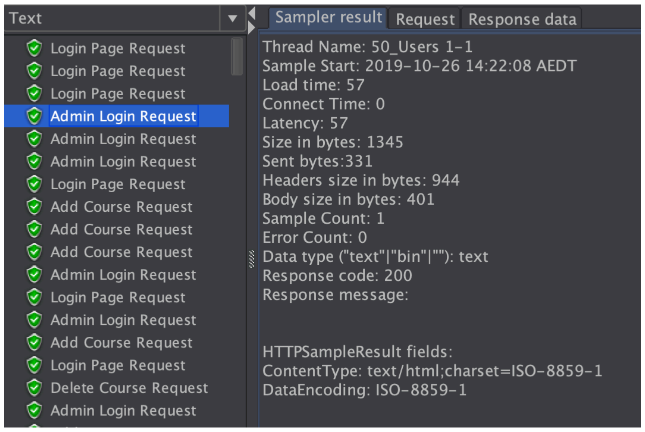
☐ Redirect Automatically ✓ Follow Redirects ✓ Use KeepAlive ☐ Use multipart/form-data ☐ Browser-compatible headers

Parameters | Body Data | Files Upload

Send Parameters With the Request:

| Name: | Value | URL Encode? | Content-Type | Include Equals? |
|---|---|---|---|---|
| userName | admin | ☐ | text/plain | ✓ |
| passWord | admin | ☐ | text/plain | ✓ |
| type | admin | ☐ | text/plain | ✓ |

Figure 16: HTTP Request to Login



Text

- Login Page Request
- Login Page Request
- Login Page Request
- Admin Login Request
- Admin Login Request
- Admin Login Request
- Login Page Request
- Add Course Request
- Add Course Request
- Add Course Request
- Admin Login Request
- Login Page Request
- Admin Login Request
- Add Course Request
- Login Page Request
- Delete Course Request
- Admin Login Request

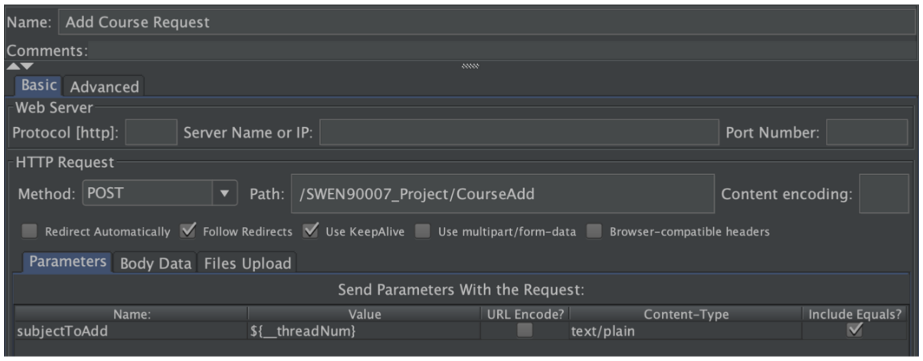
Sampler result | Request | Response data

```
Thread Name: 50_Users 1-1
Sample Start: 2019-10-26 14:22:08 AEDT
Load time: 57
Connect Time: 0
Latency: 57
Size in bytes: 1345
Sent bytes:331
Headers size in bytes: 944
Body size in bytes: 401
Sample Count: 1
Error Count: 0
Data type ("text"|"bin"|""): text
Response code: 200
Response message:


HTTPSampleResult fields:
ContentType: text/html;charset=ISO-8859-1
DataEncoding: ISO-8859-1
```

Figure 17: Response for 1 Thread logging in

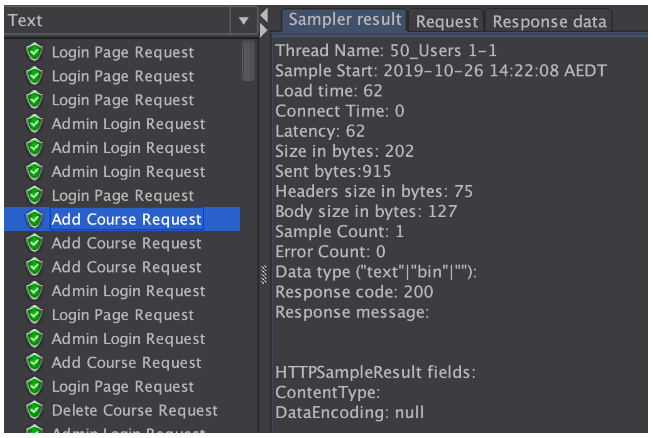
- The admin adds a list of courses to the database [Figure 18 & 19]



Name: Add Course Request
Comments:
Basic | Advanced
Web Server
Protocol [http]: | Server Name or IP: | Port Number:
HTTP Request
Method: POST | Path: /SWEN90007_Project/CourseAdd | Content encoding:
☐ Redirect Automatically ☑ Follow Redirects ☑ Use KeepAlive ☐ Use multipart/form-data ☐ Browser-compatible headers
Parameters | Body Data | Files Upload
Send Parameters With the Request:

| Name: | Value | URL Encode? | Content-Type | Include Equals? |
|---|---|---|---|---|
| subjectToAdd | ${__threadNum} | ☐ | text/plain | ✓ |

Figure 18: HTTP Request to Add Courses



```
Text
✓ Login Page Request
✓ Login Page Request
✓ Login Page Request
✓ Admin Login Request
✓ Admin Login Request
✓ Admin Login Request
✓ Login Page Request
✓ Add Course Request
✓ Add Course Request
✓ Add Course Request
✓ Admin Login Request
✓ Login Page Request
✓ Admin Login Request
✓ Add Course Request
✓ Login Page Request
✓ Delete Course Request
```

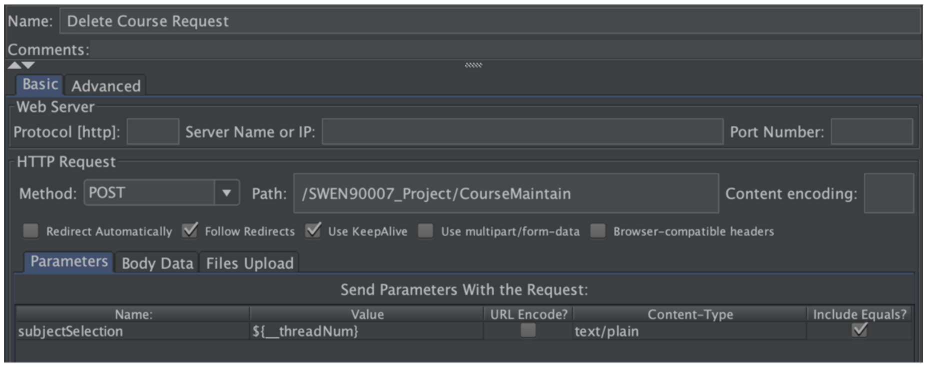
Sampler result | Request | Response data

```
Thread Name: 50_Users 1-1
Sample Start: 2019-10-26 14:22:08 AEDT
Load time: 62
Connect Time: 0
Latency: 62
Size in bytes: 202
Sent bytes:915
Headers size in bytes: 75
Body size in bytes: 127
Sample Count: 1
Error Count: 0
Data type ("text"|"bin"|""):
Response code: 200
Response message:


HTTPSampleResult fields:
ContentType:
DataEncoding: null
```

Figure 19: Response for 1 Thread Adding Courses

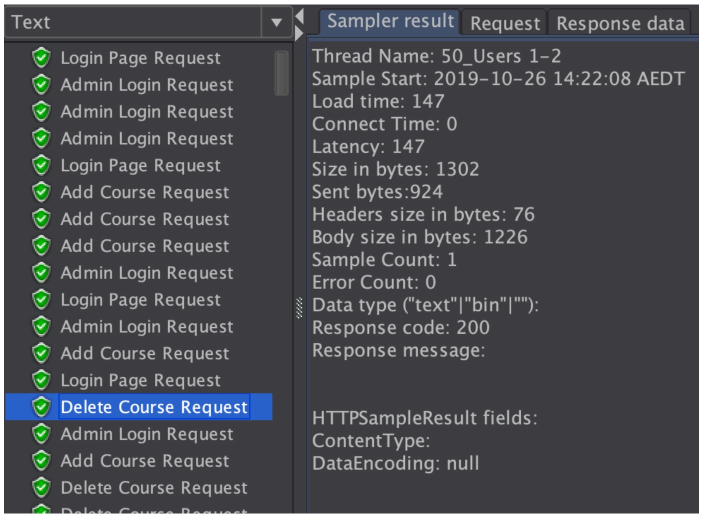
- The admin deletes the courses that were added [Figure 20 & Figure 21]



Name: Delete Course Request
Comments:
Basic | Advanced
Web Server
Protocol [http]: | Server Name or IP: | Port Number:
HTTP Request
Method: POST | Path: /SWEN90007_Project/CourseMaintain | Content encoding:
☐ Redirect Automatically ☑ Follow Redirects ☑ Use KeepAlive ☐ Use multipart/form-data ☐ Browser-compatible headers
Parameters | Body Data | Files Upload
Send Parameters With the Request:

| Name: | Value | URL Encode? | Content-Type | Include Equals? |
|---|---|---|---|---|
| subjectSelection | ${__threadNum} | ☐ | text/plain | ✓ |

Figure 20: HTTP Request to Delete Courses


Text

- Login Page Request
- Admin Login Request
- Admin Login Request
- Admin Login Request
- Login Page Request
- Add Course Request
- Add Course Request
- Add Course Request
- Admin Login Request
- Login Page Request
- Admin Login Request
- Add Course Request
- Login Page Request
- Delete Course Request
- Admin Login Request
- Add Course Request
- Delete Course Request

Sampler result | Request | Response data

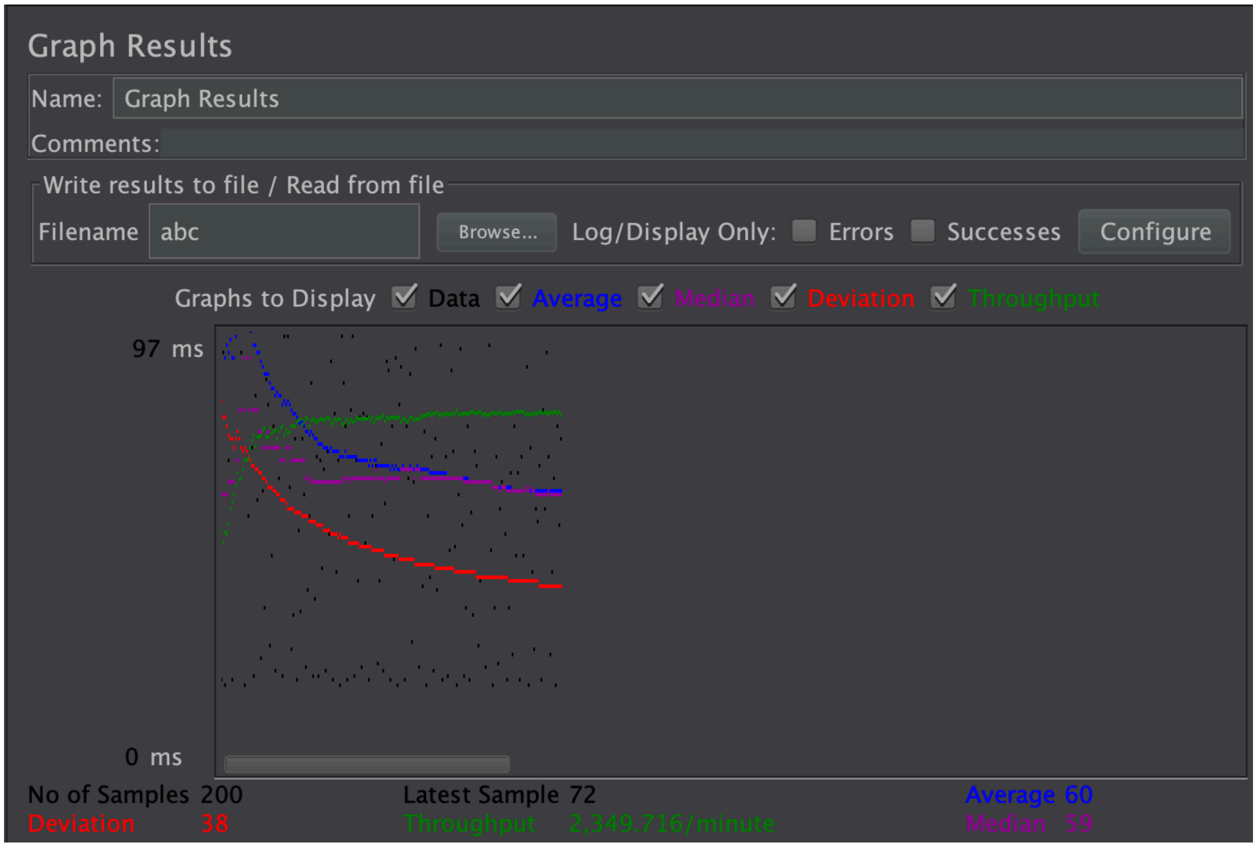
```
Thread Name: 50_Users 1-2
Sample Start: 2019-10-26 14:22:08 AEDT
Load time: 147
Connect Time: 0
Latency: 147
Size in bytes: 1302
Sent bytes:924
Headers size in bytes: 76
Body size in bytes: 1226
Sample Count: 1
Error Count: 0
Data type ("text"|"bin"|""):
Response code: 200
Response message:


HTTPSampleResult fields:
ContentType:
DataEncoding: null
```

Figure 21: Response for 1 Thread Deleting Courses

## Test Results

The final graph from the Jmeter Testing is attached below (Figure 22)


Graph Results

Name: Graph Results

Comments:

Write results to file / Read from file

Filename: abc

Browse...
Log/Display Only: Errors Successes
Configure
Graphs to Display ✓ Data ✓ Average ✓ Median ✓ Deviation ✓ Throughput
97 ms
0 ms

```
No of Samples 200        Latest Sample 72                          Average 60
Deviation     38         Throughput    2,349.716/minute            Median  59
```

Figure 22: Testing Results

The tests created show that the server can handle concurrent load on CRUD without any problems, since response latency stays under 100ms even when 50 concurrent threads are hitting the same request.

On the flipside, these tests reflect the performance of the hosting server rather than the application itself. There may be loss in performance if the allocated resources are cut down. The current infrastructure is a dedicated VM (1 vCPU, 3.75GB RAM) hosting the web and database server, located in Sydney, Australia. The low latency can also be attributed to a short geographical distance between the test and server locations.

## Performance Improvement Discussions

Multiple approaches can be taken to boost performance at the Architectural level for an enterprise system. The usage of pipelining can be done when the system is fully developed with release features, as opposed to its current state of an executable architecture. The system will be a lot more resource intensive at that level, where content loading requests can be pipelined to streamline requests and hence boost performance. This can be boosted further by delivering static content using a Content Delivery Network (CDN). The latency on localized CDNs are extremely low making it work even faster. This may not help much in a scenario where users of the system are concentrated within a single geographical area, and there it would make much more sense to host the system at a location closest to the serving location.

Another possible application of a pattern when number of users' increase is Sharding, where databases are distributed into clusters to split load and performance amongst them. One way of achieving this is to migrate from the current PostgreSQL setup to a distributed Database like the upcoming NewSQL technologies or move off relational databases and use a CouchDB NoSQL structure.

A technique that has been popular for quite a while is to use gZip compression to deliver the html files to clients. This allows a lot more data to be delivered at the same bandwidth levels, boosting end user satisfaction.

Speaking of the system as a whole, the usage of vanilla JSPs and Servlets is fairly outdated and is used majorly to maintain legacy systems, or in niche applications where the usage cannot be avoided. JSPs are inherently server side only and need to be paired with a client-side framework like JavaScript to provide any sort of client scripting. Modern Web Applications are designed in such a way that the Front End is served on the client side using JavaScript and it's reliant frameworks like AngularJS and ReactJS. These frameworks are designed to consume RESTful services from servers, decoupling the client and server side from each other.
Reactive Domain Modelling is a possible usage, where the current application design can be migrated to a relatively modern design pattern.

Another modern design implementation is the usage of frameworks like Spring and Hibernate, where Spring MVC designs have been built over the traditional Servlet APIs and Hibernate is a one stop solution to object relational mapping, handling object-relational impedance internally. These frameworks simplify the process of development by packaging a lot of the work that developers need to work on manually with traditional servlets. This also ensures that the developers do not skip important components since they are pre done in the framework.

## 7.0 Reflection

This project has gone through multiple changes and iterations over its relatively short lifespan. The first deliverable was actually done for a basic Accounting assistant software but needed to be dropped due to unclear features and roles. The project selected finally evolved into a Tutor Management System with 3 different roles. We initially started split feature implementations A and B based on roles, but decided it wasn't possible due to an unfair split.

Feature A was finally designed and built with a feature split, where this deliverable would take care of the searching and viewing of courses and tutors, while the next deliverable would handle the booking system. The initial design actually started shaping up as a transaction script since that seemed inherently simpler, but due to its scalability issues and the explicit requirements: it was re-written into the domain logic design. The Object to Relational structural patterns is extremely similar to how normalized database design works and hence was fairly straight forward to implement. On the flipside, the implementation of ORBD patterns wasn't as straight forward since it was hard to identify components where they should and shouldn't be implemented, to avoid unnecessary usage of such structures.

Feature B is when I started to work alone on the project, and the scope of features and patterns was reduced to fit a single person. I ended up using a blend of MVC and MVVC to reduce the effort behind creating separate files and structures for each view, and yet retaining the right amount of cohesion and coupling as required. For future implementations, if I had the option, I'd change the entire technology stack, and move to NodeJS backend serving RESTful APIs with a ReactJS client-side for such an application or use Spring MVC for presentation layer design if I had to stick to the Java EE suite.

When it came to implementing sessions, I decided to stick to the Server Session State patterns since the current system is enclosed within a single server and does not deal with clustering/distribution. A client session state would work better in such a scenario but would then have possible security issues. The best possible solution for a distributed scenario is to implement a hybrid session state, where sessions are maintained on both sides, but would result in additional computing power usage to ensure synchronization between session states.

The final implementations were of concurrency and security patterns. All the security patterns that I was aware of were implemented as required. The authentication and authorization enforcers with role-oriented design ensure that only the right people get access and get access only to resources that they should.

It has been a very challenging project to work on, thanks to its fairly open-ended nature, along with the challenge of working in a dysfunctional team, to working on it solo

## Appendix

**Git Repository**
https://github.com/kkkkkaran/SWEN90007project

**Deployment**
https://35.244.116.157:8443/SWEN90007_Project/

Testing:
The system is connected to a live database. The only assumption is that error handling has not been implemented. Using invalid credentials to login might cause the system to crash.
To login as student or tutor, create a new account by registering, and then login

For tutor: after creating an account, Register for courses and enter available date/time slots. Enter every new date/time slot in a new line

To login as admin: admin,admin are the credentials.