

Università degli Studi di Salerno

Corso di Ingegneria del Software

MyGym

Versione 5.1

LOGO PROGETTO



Data: 21/12/2025

Progetto: Nome Progetto

Versione: X.Y

Documento: Titolo Documento	Data: GG/MM/AAAA
-----------------------------	------------------

Coordinatore del progetto:

Nome	Matricola
Raffaele Cirillo	0512119545
Luigi Aquino	0512120291

Partecipanti:

Nome	Matricola
Raffaele Cirillo	0512119545
Luigi Aquino	0512120291
Gerardo Aquino	0512115624
Vincenzo Giordano	0512119470

Scritto da:	Raffaele Cirillo, Luigi Aquino, Gerardo Aquino, Vincenzo Giordano
-------------	---

Revision History

Data	Versione	Descrizione	Autore
02/10/2025	1	Software prenotazioni in palestra	Raffaele Cirillo, Luigi Aquino, Gerardo Aquino, Vincenzo Giordano
08/10/2025	2	Rivisitazione della parte introduttiva del progetto con aggiunta degli attori	Raffaele Cirillo, Luigi Aquino, Gerardo Aquino, Vincenzo Giordano

14/10/2025	3	Problem statement	Raffaele Cirillo, Luigi Aquino, Gerardo Aquino, Vincenzo Giordano
28/10/2025	4	Requirements Analysis Document	Raffaele Cirillo, Luigi Aquino, Gerardo Aquino, Vincenzo Giordano
11/11/2025	4.2	Dynamic model and Object model	Raffaele Cirillo, Luigi Aquino, Gerardo Aquino, Vincenzo Giordano
25/11/2025	5.0	System Design	Raffaele Cirillo, Luigi Aquino, Gerardo Aquino, Vincenzo Giordano
21/12/2025	5.1	Object Design Document	Raffaele Cirillo, Luigi Aquino, Gerardo Aquino, Vincenzo Giordano
		Ingegneria del Software	Pagina 2 di 24

Sommario

1. Object Design Trade-offs	3
2. Packages	4
3. Diagramma delle Classi	7
4. Interfaccia delle Classi	7

1. Object Design Trade-offs

1.1 Separazione delle responsabilità vs. Complessità di sviluppo

È stato scelto di adottare rigorosamente il pattern MVC adatto ad una architettura Three-Tier. Questo comporta la separazione netta tra logica di presentazione (JSP/HTML), la logica di controllo (Servlet) e la gestione dei dati (DAO/Bean).

La scelta aumenta la complessità iniziale dello sviluppo e il numero di file/classi da gestire guadagnando però in manutenibilità e modularità.

1.2 Sicurezza vs. Usabilità

Il sistema imposta un timeout della sessione utente molto breve, pari a 10 minuti di inattività sacrificando leggermente l'usabilità (l'utente è costretto a rifare il login se si allontana per poco tempo). Si massimizza però la sicurezza e l'efficienza delle risorse del server.

1.3 Prestazioni vs. Costi

Si è optato per l'utilizzo di MySQL come DBMS e Apache Tomcat come Web Server. Tramite l'uso di queste tecnologie i costi di licenza o complessità di gestione, hanno un costo minore. Grazie a questa scelta, si garantiscono comunque delle prestazioni sufficienti a supportare un buon numero di utenti simultanei richiesti dai requisiti non funzionali.

1.4 Funzionalità vs. Tempo di Rilascio (Time-to-Market)

Il team ha deciso di implementare tutte le funzionalità previste piuttosto che ridurre lo scope per anticipare la consegna. Si garantisce un prodotto finale completo di tutti i requisiti funzionali del cliente a discapito di un possibile ritardo nella consegna in modo da non rilasciare un prodotto incompleto.

2. Packages

In conformità con il pattern **MVC** e la tecnologia Java Servlet specificata nel System Design, il sistema è decomposto nei seguenti package:

- **com.mygym.model.bean**: Contiene le classi Entità (Entity Objects) che rappresentano i dati del dominio.
- **com.mygym.model.dao**: Contiene le classi per l'accesso ai dati (Data Access Objects) che interagiscono con il database MySQL.
- **com.mygym.controller**: Contiene le Servlet che gestiscono la logica applicativa e il flusso di controllo.
- **com.mygym.view**: Contiene le pagine JSP/HTML per l'interfaccia utente.

2.1 JSP relative all'account

account.jsp → Pagina che mostra all'utente il proprio account;

cambioPassword.jsp → Pagina che mostra all'utente la pagina per inserire i dati per cambiare la password;

successoRegistrazione.jsp → Pagina che mostra all'utente l'avvenuto successo della registrazione;

successoCambioPassword.jsp → Pagina che mostra all'utente l'avvenuto successo del cambio password;

adminAmministratore.jsp → Pagina che mostra le operazioni dedicate all'amministratore;

login.jsp → Pagina che mostra all'utente la pagina di login;

loginIstruttore.jsp → Pagina che mostra all'istruttore la pagina di login;

registrazione.jsp → Pagina che mostra all'utente la pagina che consente l'inserimento dei dati di registrazione.

2.2 JSP relative alle prenotazioni

prenotazione.jsp → Pagina che mostra all'utente di selezionare una data/ora specifica per un corso e confermare la prenotazione;

dashboardUtente.jsp → Pagina che mostra all'utente loggato l'elenco delle prenotazioni attive e lo storico di quelle passate;

adminPrenotazioni.jsp → Pagina che mostra le operazioni dedicate al gestore Prenotazioni.

2.3 JSP relative ai corsi

corsi.jsp → Pagina che mostra all'utente l'elenco dei corsi disponibili con filtri per tipologia e istruttore;

dettagliCorso.jsp → Pagina che mostra all'utente le informazioni specifiche di un singolo corso;

adminCorsi.jsp → Pagina che mostra le operazioni dedicate al gestore Corsi.

2.4 JSP relative agli abbonamenti

abbonamento.jsp → Pagina che mostra all'utente lo stato dell'abbonamento.

2.5 JSP generali

Index.jsp → Pagina che mostra all'utente l'home page del sito;

headerNavBar.jsp → Pagina che mostra all'utente la barra di navigazione;

footer.jsp → Pagina che mostra all'utente il footer in fondo alle pagine

2.1.1 Package GestioneAccount

UtenteBean.java → Questa classe rappresenta le informazioni di un utente;

IstruttoreBean.java → Questa classe rappresenta le informazioni di un istruttore;

AccountControl.java → Questa classe è control (servlet) si occupa di gestire le richieste web mostrando lo storico delle prenotazioni effettuate da un utente;

IstruttoreBean.java → Questa classe estende UtenteBean e rappresenta un istruttore, contiene i dati relativi ai corsi gestiti e ai feedback ricevuti;

CambioPasswordControl.java → Questa classe è control (servlet) si occupa di gestire le richieste web per modificare la password dell'utente;

GestoreIstruttoreControl.java → Questa classe è control (servlet) si occupa di gestire le richieste web per la gestione del personale;

GestoreIstruttoreModelDS.java → Questa classe contiene i metodi che permettono di effettuare le operazioni di validazione, aggiunta, rimozione e modifica dei dati dell'istruttore;

LoginModelDS.java → Questa classe contiene i metodi che permettono di effettuare le operazioni di validazione e selezione dei dati dell'utente;

LoginControl.java → Questa classe è control (servlet) si occupa di ricevere i dati di login, elaborarli e decidere se consentire o meno l'accesso all'area personale.

LoginIstruttoreControl.java → Questa classe è un control (servlet) si occupa di ricevere i dati di login, elaborarli e decidere se consentire o meno l'accesso all'area personale di un istruttore;

RegistrazioneModelDS.java → Questa classe contiene i metodi che permettono di effettuare le operazioni di visualizzazione, aggiunta, cancellazione e modifica degli account utente;

LogoutControl → Questa classe è un control (servlet) si occupa di ricevere le richieste per invalidare la sessione;

RegistrazioneControl.java → Questa classe è control (servlet) si occupa di ricevere i dati di registrazione, elaborarli e permettere la registrazione dell'utente.

2.1.2 Package gestione corsi

CorsoBean.java → Questa classe rappresenta un corso offerto dalla palestra. Contiene informazioni come nome, descrizione, data, orario, capienza massima e istruttore assegnato;

CorsoControl.java → Questa classe è un control (servlet) che gestisce le richieste web relative alla creazione, modifica, eliminazione e visualizzazione dei corsi;

CorsoModelDS.java → Questa classe contiene i metodi per l'accesso ai dati persistenti dei corsi.

2.1.3 Package gestione prenotazioni

Prenotazionebean.java → Questa classe rappresenta una prenotazione effettuata da un cliente per un determinato corso e una specifica data e ora;

PrenotazioneControl.java → Questa classe è un control (servlet) che si occupa di gestire le richieste di prenotazione, conferma e disdetta dei corsi, garantendo operazioni per evitare overbooking;

PrenotazioneModelDS.java → Questa classe contiene i metodi che permettono di salvare, aggiornare e cancellare le prenotazioni nel database.

2.1.4 Package gestione abbonamenti

AbbonamentoBean.java → Questa classe rappresenta l'abbonamento di un cliente, includendo tipologia, data di inizio, data di scadenza e stato di validità;

AbbonamentoControl.java → Questa classe è un control (servlet) che gestisce le operazioni di rinnovo e verifica della validità degli abbonamenti;

AbbonamentoModelDS.java → Questa classe contiene i metodi per la gestione persistente degli abbonamenti nel database.

2.1.5 Package notifiche

NotificaBean.java → Questa classe rappresenta una notifica inviata automaticamente dal sistema agli utenti;

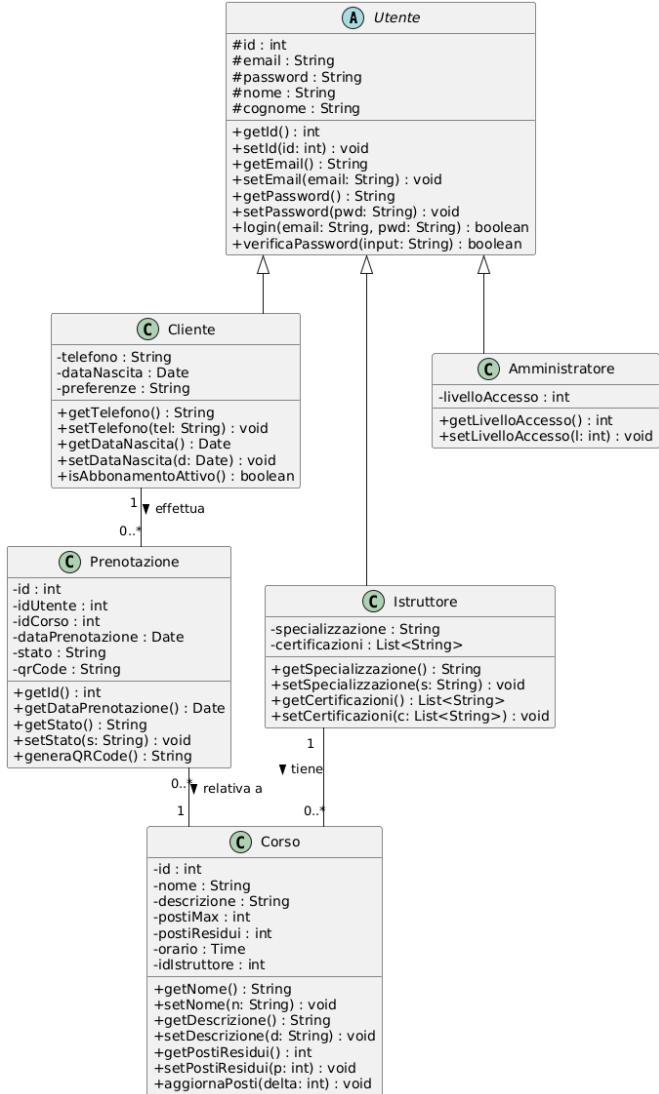
NotificaControl.java → Questa classe è un control (servlet) che si occupa di generare e inviare notifiche agli utenti in seguito a prenotazioni, modifiche o cancellazioni.

2.1.6 Package report

ReportBean.java → Questa classe rappresenta un report amministrativo contenente statistiche su corsi, presenze e abbonamenti;

ReportControl.java → Questa classe è un control (servlet) che gestisce la generazione ed esportazione dei report per l'amministratore.

3. Diagramma delle classi



4. Interfaccia delle classi

PrenotazioneModelDS

Context: PrenotazioneModelDS: doSave(prenotazione: Prenotazione): Boolean

Description: Questo metodo registra una nuova prenotazione nel database, associando un cliente a un corso specifico.

Pre-Condition: La prenotazione deve essere diversa da null, il cliente e il corso associato alla prenotazione devono esistere nel database e il corso deve avere almeno un posto disponibile.

Post-Condition: Return true se l'operazione è andata a buon fine, false altrimenti.

Context: PrenotazioneModelDS: doDelete(idPrenotazione:int): Boolean

Descrizione: Questo metodo consente di eliminare una prenotazione esistente dal database, a seguito di una disdetta da parte del cliente.

Pre-Condizione: idPrenotazione deve essere valido e associato a una prenotazione esistente nel database.

Post-Condizione: Return true se la prenotazione viene rimossa dalla tabella Prenotazione, false altrimenti.

Context: PrenotazioneModelDS: doRetrieveByUtente(email: String): Collection<Prenotazione>

Descrizione: Questo metodo consente di recuperare tutte le prenotazioni associate a un determinato utente identificato tramite e-mail.

Pre-Condizione: Il parametro e-mail deve essere diverso da null, l'e-mail deve essere associata a un utente registrato nel sistema.

Post-Condizione: Il metodo restituisce una collezione contenente tutte le prenotazioni associate all'utente.

Se l'utente non ha prenotazioni, viene restituita una collezione vuota.

Context: PrenotazioneModelDS: doRetrieveByCorso(idCorso:int): Collection<Prenotazione>

Descrizione: Questo metodo consente di recuperare tutte le prenotazioni associate a uno specifico corso.

Pre-Condizione: idCorso deve essere valido e associato a un corso esistente nel database.

Post-Condizione: Il metodo restituisce una collezione contenente tutte le prenotazioni del corso specificato. Se il corso non ha prenotazioni, viene restituita una collezione vuota.

Classe: CorsoModelDS

Context: CorsoModelDS: doUpdate(corso: Corso): Boolean

Descrizione: Questo metodo aggiorna le informazioni di un corso.

Pre-Condizione: Il corso deve essere diverso da null e il corso deve essere esistente all'interno del database.

Post-Condizione: Return true se i dati all'interno del corso vengono aggiornati nella tabella Corso, false altrimenti.

Context: CorsoModelDS: doSave(corso: Corso): Boolean

Descrizione: Questo metodo consente di inserire un nuovo corso all'interno del database.

Pre-Condizione: L'oggetto corso deve essere diverso da null, i dati del corso devono essere validi (nome, data, orario, capienza).

Post-Condizione: Return true se il corso viene inserito nella tabella Corso, false altrimenti.

Context: CorsoModelDS: doDelete(idCorso:int): Boolean

Descrizione: Questo metodo consente di eliminare un corso esistente dal database.

Pre-Condizione: idCorso deve essere valido e associato a un corso presente nel database.

Post-Condizione: Return true se il corso viene eliminato dalla tabella Corso, false altrimenti.

Context: CorsoModelDS: doRetrieveAll(order: String): Collection<Corso>

Descrizione: Questo metodo consente di recuperare l'elenco di tutti i corsi presenti nel database, ordinandoli secondo il criterio specificato.

Pre-Condizione: Il parametro order deve essere diverso da null e contenere un valore valido.

Post-Condizione: Il metodo restituisce una collezione contenente tutti i corsi ordinati secondo il criterio indicato.

Context: CorsoModelDS: doUpdateDisponibilita (idCorso:int, posti:int): Boolean

Descrizione: Questo metodo consente di aggiornare il numero di posti disponibili per un determinato corso.

Pre-Condizione: idCorso deve essere valido, posti deve essere un valore maggiore o uguale a zero.

Post-Condizione: Return true se il numero di posti disponibili viene aggiornato nella tabella Corso, false altrimenti.

Classe: AccountModelDS

Context: AccountModel: doSave(utente: Utente)

Descrizione: Questo metodo consente di aggiungere un nuovo Utente all'interno della tabella Cliente del database.

Pre-Condizione: L'utente non deve essere diverso da null e l'e-mail deve essere diversa da tutte le altre e-mail appartenenti agli utenti già presenti nella tabella Clienti.

Post-Condizione: L'utente è visibile nella tabella Cliente e la sua e-mail è univoco.

Context: AccountModel: doDelete(utente: Utente): Boolean

Descrizione: Questo metodo consente di rimuovere un utente all'interno della tabella clienti del database restituendo true se è andato a buon fine, false altrimenti.

Pre-Condizione: L'utente deve essere diverso da null e la sua e-mail deve essere presente all'interno di una tupla nella tabella Clienti.

Post-Condizione: Return true se la rimozione all'interno della tabella Cliente è andata a buon fine e rimuovendo una tupla all'interno della tabella Clienti, false altrimenti.

Context: AccountModel: doUpdate(utente: Utente): Boolean

Descrizione: Questo metodo consente di aggiornare i dati dell'utente all'interno della tabella Clienti del database restituendo true se è andato a buon fine, false altrimenti.

Pre-Condizione: L'utente deve essere diverso da null e l'e-mail dell'utente deve essere presente all'interno di una tupla della tabella Utente del database;

Post-Condizione: Return true se l'operazione ha successo e i dati dell'utente risultano aggiornati nella tabella Utente, false altrimenti.

Context: AccountModelDS: doUpdatePassword(utente: Utente): Boolean

Descrizione: Questo metodo consente di modificare la password associata a un utente registrato all'interno della tabella Utente del database, restituendo true se l'operazione è andata a buon fine, false altrimenti.

Pre-Condizione: L'utente deve essere diverso da null e l'e-mail dell'utente deve essere presente all'interno di una tupla della tabella Utente.

Post-Condizione: Return true se l'operazione ha successo ovvero la password dell'utente viene aggiornata nel database, false altrimenti.

Context: AccountModelDS: doRetrieveAll(order: String): Collection<Utente>

Descrizione: Questo metodo consente di recuperare l'insieme di tutti gli utenti all'interno della tabella Clienti del database:

Pre-Condizione:

Post-Condizione: Return dell'insieme di tutte le tuple all'interno della tabella Clienti.

Context: AccountModel: doRetrieveByKey(email: String): Utente

Descrizione: Questo metodo consente di ottenere l'utente all'interno della tabella Clienti del database con l'e-mail specificata.

Pre-Condizione: L'e-mail deve essere diverso da null e deve essere presente all'interno di una tupla nella tabella Clienti.

Post-Condizione: Return dell'utente con e-mail specificato dall'interno della tabella Clienti.

Classe: LoginModelDS

Context: LoginModelDS: doRetrieveByKeyIstruttore(code: String)

Descrizione: Questo metodo consente di ottenere l'istruttore all'interno della tabella Istruttori del database con e-mail specificata.

Pre-Condizione: Istruttore deve essere diverso da null e l'istruttore con e-mail cercata deve essere presente all'interno di una tupla nella tabella Istruttori.

Post-Condizione: Return del personale con e-mail specificata dall'interno della tabella Istruttore.

Context: LoginModelDS: doRetrieveByKey(code: String)

Descrizione: Questo metodo consente di ottenere il cliente all'interno della tabella Cliente del database con e-mail specificata.

Pre-Condizione: Il cliente deve essere diverso da null e il cliente con e-mail cercata deve essere presente all'interno di una tupla nella tabella Cliente.

Post-Condizione: Return del Cliente con e-mail specificato dall'interno della tabella Cliente.

Classe: GestoreIstruttoreModelDS

Context: GestoreIstruttoreModelDS: stampaTuttiGliIstruttori(): Collection<IstruttoreBean>

Descrizione: Questo metodo consente di ottenere l'insieme del personale all'interno della tabella Istruttore del database;

Pre-Condizione:

Post-Condizione: Return dell'insieme di tutte le tuple all'interno della tabella Istruttore.

Context: GestorePersonaleModelDS: doRetriveByKey(code: String): Istruttore

Descrizione: Questo metodo consente di ottenere l'istruttore all'interno della tabella Istruttore del database con e-mail specificata.

Pre-Condizione: code diverso da null e code deve essere presente all'interno di una tupla nella tabella Istruttore.

Post-Condizione: Return dell'istruttore con e-mail specificata all'interno della tabella Istruttore.