ChatGPT

Architectural and Design Flaws

Layered Architecture & Coupling: The API is organized into handlers (controllers), services (business logic), and repositories (data access), which is a solid separation of concerns. However, there are some coupling issues. For example, the `CourseService` calls other services to validate existence (university, faculty, etc.) and to fetch related data (faculty/professor names) for responses [1] [2]. This cross-service calling (instead of using repositories or joins) can lead to inefficiencies and tight coupling. In the `CourseService.Search` implementation, each course is mapped to a DTO by making separate service calls to get faculty and professor details [2] [3], causing **N+1 query** behavior and unnecessary coupling between services. A better design would preload or join related data at the repository level (indeed, the code attempts a `Joins("Faculty").Joins("Professor")` in the query [4] but then ignores these joined results), or maintain a clearer separation where services don't call other services just to enrich data for response.

Transaction Boundaries: The repository layer appropriately uses transactions for multi-step operations (e.g. creating a course and its time slots in one transaction [5], or deleting a course with its child records in one go [6] [7]). However, there is a flaw in the batch creation logic: after inserting courses in a loop, the code fetches *all* courses from the database (no filter) and returns them [8]. This is likely a bug – it should return only the newly created courses. This could lead to returning massive datasets unintentionally and indicates insufficient testing of that path.

Business Logic in Controllers vs Services: The handlers mostly delegate to services, but some input handling is inconsistent. For example, in user registration, the handler parses UUID strings to `uuid.UUID` and packages a service request [9] [10], but does not verify that the referenced university or faculty actually exist (no call to `UniversityService` or `FacultyService` here). The service then attempts to create the user and will fail only when the database FK constraint is violated. This is a design oversight: ideally the service should validate foreign keys (or the handler should call `universityService.Get` as is done elsewhere) to return a controlled "not found" error instead of a generic DB error.

Error Handling & Propagation: A custom `errors` package defines sentinel errors (e.g. `ErrNotFound`, `ErrConflict`, `ErrInvalid` etc.) [11] and helper constructors, which is good for consistent error semantics. Services often wrap lower errors or return these sentinels so that handlers can decide HTTP status codes. However, this is not used uniformly. For example, in `AuthService.Register`, the code checks for existing user and returns a plain error with a message `"email or student ID already exists"` [12] instead of using `errors.NewConflictError`. The auth handler then **string-compares** that error message to set a 409 Conflict [13] – a brittle approach. Other parts of the code do use `errors.NewConflictError` (e.g. course code conflict [14] or faculty short code conflict [15]), which handlers catch via `errors.Is`. This inconsistency is a code smell. It would be better to consistently use the `errors` package for conflicts and validation issues, and perhaps define error types for cases like "invalid credentials" instead of using raw strings.

Mixed Responsibility in Services: Some services are doing more than just core logic – for instance, `AuthService` handles sending emails (verification, password reset) synchronously [16]. While this is acceptable for a smaller system, in a larger system one might offload email sending to a background job or at least not block the HTTP request on email success (the code currently logs an error if email fails but proceeds anyway [17]). Similarly, the `AuthService.Refresh` (not fully shown) presumably checks a refresh token in the database. It should also mark tokens as used or revoked to prevent reuse, but there is a `revoked` field in the DB schema that doesn't appear to be utilized [18]. These indicate areas where the design could be hardened (e.g. implementing refresh token rotation and revocation fully, and decoupling email/side-effects from the request-response cycle).

Endpoint Naming and RESTfulness

Consistency in REST Paths: Most endpoints follow RESTful conventions with plural nouns and meaningful sub-routes. For example, `/v1/admin/universities` is used to create and list universities, and `/v1/admin/universities/{id}/faculties` lists faculties of a university [19]. However, there are some inconsistencies: - The route for getting courses by faculty is registered as `/v1/admin/faculties/courses/:facultyID` [20], whereas a more conventional approach would be `/v1/admin/faculties/{facultyID}/courses`. In fact, the handler's documentation assumes the latter [21], but the actual route is different. This mismatch could confuse developers and breaks RESTful hierarchy. - Similarly, the course search endpoint is documented as `/v1/courses [get]` [22] (implying a general search), but in code it's mounted under the admin group [23]. So only admins can use it, even though search/filtering might be useful for regular users. This may be intentional, but if not, it's a design issue. The documentation and implementation should be aligned – likely by moving `GET /courses` out of the admin group for all users, or adjusting the docs.

Noun vs Verb: The API uses mostly nouns for resources. One exception is the user courses selection: they chose endpoints like `/v1/courses/select` (POST to add a course) and `/v1/courses/select/{courseId}` (DELETE to remove) [24]. These could be made more RESTful. For instance, adding a course to a user's plan could be `POST /v1/user/courses` with the course ID in the body, and removal `DELETE /v1/user/courses/{id}`. The current approach with the word "select" in the path is more RPC-style. However, this is a minor stylistic issue; it does convey the action.

Plural vs Singular: Mostly plural nouns are used (`universities`, `faculties`, etc.). One slight oddity is `/universities/{id}/faculty/{short_code}` (singular "faculty") [25]. This is effectively fetching one faculty by a compound key (university + short code). It works, but for consistency they might use `/universities/{id}/faculties/{short_code}`. This inconsistency doesn't break functionality but could be standardized.

Versioning: The API is versioned under `/v1`, which is good for future evolution. They also expose Swagger UI under `/v1/swagger/…` [26]. One thing to ensure is that any future changes are reflected under incremented version prefixes to avoid breaking clients.

Security Best Practices

Authentication & Authorization: The API uses JWT for auth and a middleware to protect routes. The `JWTMiddleware` correctly checks the `Authorization: Bearer` header and validates the token [27] [28]. It also handles expired tokens explicitly, returning a 401 with “Token expired” [29]. Authenticated requests put `userID` and `userScopes` in the context for handlers to use [30]. For admin-only routes, an `AdminMiddleware` checks that the JWT scopes contain the required `"admin-dashboard"` scope [31]. This is a good approach to authorization – using token scopes – assuming the token issuance is secure. The code sets that scope if `user.IsAdmin` on login [32], so it’s robust as long as the signing key remains secret.

Input Validation: There is basic payload validation via Gin’s binding tags on DTOs. For example, `CreateCourseDTO` requires many fields and even enforces certain formats (UUIDs, `oneof` for gender restriction, etc.) [33] [34]. This helps catch malformed input. In many handlers, they also manually validate path params (e.g. UUID format) and return 400 if invalid [35] [36]. One gap, as mentioned, is that certain dependencies aren’t validated for existence (registering with a non-existent faculty ID). Also, some error messages leak internal detail: in registration, `ShouldBindJSON` errors are returned directly with `err.Error()` [37], which might include messages about JSON decoding/validation. It would be better to standardize these to a user-friendly “Invalid request format” as done elsewhere [38].

Sensitive Data Handling: Passwords are properly hashed using bcrypt before storing [39]. The JWT contains user ID and scopes, but no sensitive personal data. Refresh tokens are stored server-side, which is more secure than only relying on JWT expiry. However, the refresh token implementation could be improved: the `RefreshToken` model has a `revoked` flag [40], but the code doesn’t mark tokens as revoked on logout – instead, it likely deletes them (the `Logout` service likely queries and deletes or sets revoked). Ensuring that refresh tokens are invalidated on logout and not usable twice is important (e.g., implement *rotating refresh tokens* and use the `revoked` field).

Rate Limiting and Brute-Force Protection: There is no evidence of rate limiting on login or other sensitive endpoints. The `AuthService.Login` logs failed attempts (user not found, wrong password, etc.) [41] [42] but does not throttle them. In a production scenario, adding rate limiting (e.g., using middleware or an API gateway) for login and other auth endpoints would prevent brute force attacks. Similarly, the password reset flow does not appear protected against abuse – it returns immediately with a success message (not revealing if email exists) [43], which is good, but sending emails could be abused; rate limiting those requests per IP would be wise.

CORS Configuration: The CORS middleware is configured to allow the front-end origin and credentials [44]. This is appropriate for a first-party SPA. Just ensure that in production the `cfg.FrontendURL` and any needed domains are correctly set, and that `AllowCredentials: true` is actually needed (it likely is, since JWT is in auth header and refresh tokens are HTTP-only cookies or stored in client – if using cookies, they’d need credentials; but here JWT is header-based, so credentials may not be strictly required).

Error Handling

HTTP Status Codes & Messages: In general, the handlers map errors to appropriate HTTP codes. Not found errors from services translate to 404, validation issues to 400, conflicts to 409, etc. For example, in `UniversityHandler.Update`, a not-found error from service yields a 404 [45], and a validation error yields 400 with the error message [46]. The consistency of using `errors.ErrNotFound`, `ErrInvalid`, etc., helps maintain this mapping. One issue is that some handlers don't catch all cases: the `CourseHandler.Search` doesn't explicitly handle the case where filters are invalid. The service's `validateSearchFilters` might return a `NewValidationError` (underlying `ErrInvalid`) [47], but the handler treats any error from search as 500 [48]. This means if an invalid UUID filter is given, the user would see a generic "Internal server error" instead of a 400 message about bad parameters. This is likely an oversight.

Logging and Error Detail: The application uses Uber Zap for logging. Errors are generally logged at the service level (with context like service name, operation, IDs) and at the handler level for unexpected failures. This is good for debugging. However, some error responses to clients are very generic ("Internal server error" for most unhandled exceptions) – which is secure (not leaking internals) but could harm user experience if too frequent. Perhaps including a request ID or logging correlation id would help trace issues in logs without exposing details to users.

Redundancy in Error Handling: The code often checks for existence *twice*: e.g. `FacultyService.Create` calls `universityService.Get` to ensure the university exists, and also checks for duplicate short code in that university [49] [15]. But then, when calling `facultyRepo.Create`, the database might error (e.g. if a unique index existed on (university_id, short_code), which currently it doesn't). The pattern of checking and then inserting is okay, but it introduces a race condition window. A more robust approach is to rely on a DB unique constraint and catch the error – since this is a low-concurrency system, it's not a big issue, but something to consider for enterprise robustness. Additionally, every handler repeats the same `uuid.Parse` and error mapping logic. This could be refactored (for example, a middleware to validate and attach UUID params to context) to reduce repetition, but that's a minor maintainability improvement.

Documentation and Developer Experience

Inline Documentation: The code is annotated with Swagger (using swaggo comments) for each endpoint, including summaries, descriptions, parameter docs, and response schemas. This is excellent for developer experience, as it enables auto-generated API docs. The presence of `@Security BearerAuth` on protected endpoints and explicit tagging of error responses shows attention to detail. One problem is that some Swagger annotations are out of sync with the code: e.g., the `CourseHandler.Search` is documented under `/courses [get]` [22] but the actual route is `/admin/courses` as noted. Such mismatches could confuse users reading the docs. A thorough review of the generated Swagger output against the implemented routes is needed to fix any discrepancies.

Repository README: The README provides a high-level overview of the system (Termustat) and even an architecture diagram [50]. It explicitly warns that the project is under refactor and not production-ready [51], which might explain some of the rough edges we see. It's good that the README includes setup instructions

(Docker, env config) and a pointer to the Swagger UI [52]. For developer experience, one might additionally include example curl commands or a Postman collection, but the available info is likely sufficient given the API is self-documented via Swagger.

Error Responses and API Feedback: The API returns error messages in a simple JSON format (`{"error": "message"}`). While functional, the team could consider a more structured error format (e.g., an error code, field errors for validation, etc.) for clients to handle programmatically. Also, some success responses use a message string (e.g., delete endpoints return `{"message": "… deleted successfully"}` [53] [54]). It might be more RESTful to return 204 No Content on deletions. Consistency in such responses will improve the DX (currently some deletes yield JSON message, others might not; in our scan, University and Course deletes do return a JSON message).

Testing: The presence of some `_test.go` files (saw references to `auth_test.go`) suggests there are unit tests, at least for auth. It's not clear how extensive they are. If this is a critical system, more tests (especially for the complex scheduling logic and conflict validation) would be beneficial. From a DX perspective, having those tests also serves as examples of usage. If tests are lacking, adding them would be a key maintainability improvement.

Code Duplication and Inconsistencies

Repeat Patterns: Many handlers and services have very similar code blocks for different entities. For instance, nearly every entity service has a `Create -> repo.Create -> return DTO`, `Update -> repo.Find + repo.Update`, etc., and the handlers do the same pattern of binding JSON, calling service, then a switch on error for 404/400/500. This yields a lot of duplicated code. While this repetition is somewhat expected in CRUD boilerplate, there are opportunities to factor common logic: - **Validation:** The field checks for empty strings or nil booleans are repeated in multiple services (University, Faculty, Course, etc.). Consider using struct tags with a validation library (Gin's binding does some, but custom rules could be added) or a shared helper to validate required fields. The code currently uses `errors.NewValidationError("field")` in each service function for each field [55] [56] – that's a lot of repetition. - **Mapping to DTOs:** Each service builds its response DTO manually. They have some helper mappers (`mapCourseToResponse`, etc.), but they are defined per service (e.g., `mapFacultyToResponse`, `mapCourseToResponse` in their respective files). These could be consolidated or generated. Notably, there's an inconsistency: `CourseService.mapCourseToResponse` does *not* include faculty/professor names [57] [58], whereas `mapCourseToDTO` (used in Search) does fetch names [2] [3]. This inconsistency means some API responses include related names (search results) while others (normal GET) do not, unless the frontend always calls an additional endpoint to get names. Standardizing the response fields (and possibly always including the related names to improve usability) would help. It might be better to include related names in all Course responses to avoid extra client calls – the DB join and mapping logic could be used for all gets, not just search.

Logic Bug (Inconsistency): A clear inconsistency is in `CourseService.GetAllByFaculty` – it mistakenly calls the repository's `FindAllBySemester` with a facultyID [59]. The repository has a proper `FindAllByFaculty` method [60]. This is likely a copy-paste error and means the "get courses by faculty" endpoint will actually return courses by semester (or none at all, since it's using a faculty ID as a semester ID filter). This kind of bug indicates lack of integration testing for that endpoint. It should be fixed by calling the correct repo method. Keeping the code DRY (don't repeat constants or switch blocks) can reduce such mistakes; e.g., if the route handler took a generic list function with filter type.

Admin User vs Regular User Handling: The system has an `AdminUserService` (for managing admin accounts) and separates admin routes under `/admin`. However, the code treats `AdminUserService` and `UserService` somewhat confusingly. For example, `UserCourseService` uses an `AdminUserService` reference named `userService` to fetch a user's gender in order to validate gender restrictions [61]. Using an admin service to get a normal user is odd naming. Likely `AdminUserService.Get(id)` actually returns any user (admin or not) by ID. This dual use can be clarified by renaming or consolidating user services (perhaps the admin user service is just a user service with admin capabilities). It's a small internal inconsistency that could affect maintainability if roles expand.

Scalability and Performance

Database Access Patterns: The use of Gorm as an ORM simplifies data access, but caution is needed for performance: - Many queries load related data with `Preload`, which is good (e.g., loading course times with courses in one go [62]). However, as noted, some features (like search) do not preload `CourseTimes`, so those would require additional calls if the client needs that data. - The mapping of courses in search results is doing a lot of per-record processing [63]. For large result sets, this could be slow. Also, the search query itself joins the `Faculty` and `Professor` tables without selecting specific fields [4]. Depending on Gorm's behavior, this might fetch entire faculty and professor records for each course. That could be thousands of extra columns if, say, 100 courses are returned. A better approach would be to select only needed fields or use simpler joins. Since the code ultimately *didn't even use* the joined data (it re-fetches via services), the joins only add overhead with no benefit – a clear performance bug. Removing or properly utilizing those joins is important.

Pagination and Limits: None of the list endpoints appear to implement pagination or result limiting. `GetAll` for universities, courses, etc., will return the full table contents [64] [65]. This is fine for now (perhaps these lists are small in the target usage), but as data grows it will be a scalability bottleneck. Implementing pagination (e.g., `?page=` or `?limit=` params) and/or at least default limits (e.g., return first 100 by default) would make the API more robust for large data sets. The `CourseSearchFilters` has no pagination either – a search that matches many courses will send them all in one go. Consider using Gorm's `.Limit()` and `.Offset()` for these, and returning a `Total` count as done in `CourseListResponse` (there's a `Total` field in the DTO [66] that isn't actually populated anywhere we saw).

Compute-Heavy Operations: The most "compute-heavy" logic in this API relates to schedule conflict checking. The `ValidateTimeConflicts` function fetches all of a user's courses in a semester and then compares time slots pairwise [67]. This is O(n) for the number of current courses the student has – typically small (5-10). This is fine. However, it calls `courseService.Get` for each existing course in the loop [67]. Each of those calls runs a DB query. So conflict checking one new course against 5 existing courses results in 6 DB calls (one for the new course, five for each existing). This could be optimized by fetching all needed course times in one query (e.g., join user_courses -> courses -> course_times in a single SQL). Given the small numbers, this is not a pressing performance issue, but it's a scalability consideration (e.g., if a user could somehow have dozens of courses, it would scale poorly). Caching course time data in memory for the duration of the check could also be an approach (the service could accept the new course's times as input rather than its ID to avoid re-fetching it, since the handler originally had the course or could retrieve it once).

Concurrency and Connections: The database connection pool is configured with a max of 50 open connections [68]. This should be sufficient for the scale of a university course scheduler (unless there are hundreds of concurrent requests). If deploying to production, monitor connection usage; if using an autoscaling environment, ensure idle conns are trimmed. There is no explicit use of context cancellation for requests (Gin uses context under the hood, but in Gorm calls it's not leveraged), so long queries won't automatically cancel if the client disconnects – not a huge issue but something to keep in mind.

Use of Design Patterns and Separation of Concerns

Repository Pattern: The code nicely encapsulates data operations in repository structs (e.g. `CourseRepository` with methods for CRUD and specific queries [69]). This separation makes it easier to swap out the ORM or to mock the database in tests. The pattern is well-used with clear methods like `FindByUniversityAndCode`, `ExistsByName`, etc., implementing logic once and reusing it in services. One improvement could be to handle *all* relevant logic in the repository (for instance, the unique checks for conflicts could be done via DB queries or constraints rather than in the service). But overall, the pattern is applied consistently.

Service Layer: The services implement business rules (e.g., trimming strings, enforcing uniqueness, orchestrating multiple repository calls). They generally do not return raw model objects but rather DTOs, which is good for separation – the controller doesn't need to know about internal model details. The `UserCourseService` is a highlight: it encapsulates complex rules like checking capacity, gender, conflicts, etc., and keeps the handler very thin (the handler just calls `AddCourse` and returns 200 or error). This is a good use of the Service pattern to centralize rules.

MVC and Beyond: The project overall follows an MVC-like structure (without a view, since it's an API). One thing to note is the **Engine** module mentioned in the README: it parses external data into courses. The API's `CreateFromEngine` function suggests a design where the engine produces data and the API reuses its `Create` logic [70] [71]. Currently, `CreateFromEngine` simply repackages a CourseEngineDTO into the regular Create DTO and calls `Create` [72] [73]. This is fine, but it hints at a possible design improvement: if engine and API share code, perhaps the engine could call the service directly or the service could accept engine input more directly (to avoid constructing intermediate DTOs). However, this is an acceptable separation (engine as a distinct context and API service as a facade).

Middleware: The use of Gin middleware for JWT and Admin check is appropriate. They could consider adding a global recovery or error-transforming middleware. Right now, the handlers themselves catch errors and write JSON. An alternative design would be to have services return structured errors (with codes), then one middleware in Gin that translates those to JSON HTTP responses uniformly. This would reduce repetition in handlers. Given Gin's support for middleware, this could clean up controllers significantly. It's a design preference – the current approach works but leads to a lot of similar code in each handler.

Extensibility: The design allows adding new entities fairly easily by following the existing patterns. For instance, if we needed a new resource (say, “departments”), we’d create a model, repository, service, handler, and plug it into routes. The consistency in structure helps. One thing to be cautious of is the **AdminUserService vs normal user** split – currently “AdminUser” is basically the same model as “User” (just with `IsAdmin` true). There might have been a thought to separate admin accounts, but they ended up using one table with a flag. The design pattern here could be simplified: one UserService with methods to manage both admins and regular users, rather than two service types. The separation into an `AdminUserService` interface might be over-engineering unless they planned different behaviors. This is more about simplicity and avoiding confusion in the codebase.

Potential Features and Enhancements

Finally, beyond code improvements, here are some **features to consider adding** to improve the API’s functionality and usability:

- **Pagination & Filtering**: As noted, implementing pagination on list endpoints is crucial for usability when data grows. For example, `GET /v1/admin/courses` (search) could accept `page` and `page_size` and return results with a total count. This prevents overload on client and server when thousands of courses exist. Similarly, allow filtering on `GET /v1/admin/professors` or adding a `GET /v1/admin/professors` (currently missing) with query params for name, etc. Right now an admin can get professors only by university or ID [74]. A general list with search by name would be a nice addition.

- **User-Facing Course Search**: Provide non-admin users an endpoint to search or list courses for their university/semester. The front-end course planner would benefit from an API to fetch available courses. Perhaps `GET /v1/courses?university_id=X&semester_id=Y[&faculty_id=Z]` that returns courses (restricted to those parameters). This could use the existing search logic but scoped to the user’s context. Right now it seems the design assumes the Engine pre-populates courses or the front-end has them, but an API endpoint would improve flexibility.

- **Bulk Actions and Imports**: There is a `BatchCreateCourses` in the service and a corresponding DTO [75], but no exposed endpoint for it. Exposing an admin endpoint like `POST /v1/admin/courses/batch` to upload multiple courses (e.g., via a JSON array or even a CSV/Excel file parsed by the client) could speed up data entry. Additionally, an endpoint to trigger the Engine module to import data from a Golestan export file could be valuable: e.g., `POST /v1/admin/import` with a file upload, which the Engine parses and then feeds into the CourseService.Create logic. This automates the “Data Import” feature mentioned in the README [76].

- **Enhanced Conflict Feedback**: The `ValidateTimeConflicts` currently returns a generic error “time conflict with course: X” [77]. A feature improvement would be to return *which times* conflict (e.g., “conflict with course X on Monday 10:00”) so the user knows exactly what to adjust. This could be done by including conflict details in the error response or by a separate endpoint that returns all conflicts among a user’s selected courses (not just checking one). For instance, `GET /v1/courses/selected/conflicts` for a user could return a list of conflict pairs or an empty list if none. That would enhance usability for the student planning their schedule.

- **User Profile Management**: Currently, regular users can register and then only have the `/me` endpoint to get their info [78]. It would be useful to allow users to update their profile – e.g., change name or password. An endpoint like `PUT /v1/user` to update profile (with proper auth) and `PUT /v1/user/password` to change password (requiring current password) would round out the auth functionality. Right now, password reset via email is the only way to change a password, which is not always ideal. Adding these would improve user experience.

- **Email Verification Resend & Expiry**: The system sends a verification email on register. A nice feature is an endpoint to re-send verification if a user didn't get the email, e.g., `POST /v1/auth/verify-email/resend` with their email. Also, perhaps allow unverified users to trigger a new verification if the old token expired. This requires tracking verification attempts, but would make the account verification process smoother. Currently, if the token expires in 24h, the only way is probably to register again or an admin to manually verify them in DB.

- **Administrative Actions & Audit**: For admin users, additional features like viewing audit logs of user activity or listing unverified users might be helpful. For example, an admin endpoint `/v1/admin/users?verified=false` to list who hasn't verified email, so they can intervene or manually verify if needed. Also, adding creation of new admin users (as of now, admin user creation isn't exposed via API – possibly one admin exists by migration). A `POST /v1/admin/users` to invite or create another admin (perhaps sending them an activation email) could be useful for delegation.

- **Rate Limiting / Security Features**: Implementing rate limiting (perhaps via a proxy or middleware) as a feature isn't directly an API endpoint, but is an improvement. Also, features like account lockout after X failed login attempts (with an unlock via email) could be considered in the auth service to enhance security. These would prevent brute force attacks on login beyond just rate limiting.

- **Caching Frequently Used Data**: To improve performance, consider caching certain read-mostly data. For example, the list of universities or faculties doesn't change often but is used in many operations (registration, course filtering). A simple in-memory cache (or using an external cache like Redis) for `GetAllUniversities` or `GetAllFaculties` could reduce DB load and latency for those calls. Since they are small tables, this is low-hanging fruit. Just ensure cache invalidation on create/update/delete of those resources (which are relatively infrequent).

- **Improved Swagger Docs & Client SDK**: As a usability feature, ensure the Swagger documentation is complete (include models examples, ensure all routes documented correctly) and consider using it to generate client SDKs (e.g., a TypeScript API client for the frontend). This isn't an API feature per se, but it greatly improves developer experience for anyone integrating with the API.

Each of these suggestions comes with technical rationale: they either improve performance, security, or the usefulness of the API to end-users and admins. Implementing them would make Termustat's API more robust and user-friendly, especially as it moves from refactoring to a production-ready service.

---

1 2 3 14 47 57 58 59 63 70 71 72 73 **github.com**
https://github.com/ArmanJR/termustat/blob/0d05ced37a29244cff7652025375ae34bc954345/api/services/course.go

4 5 6 7 8 60 62 69 **github.com**
https://github.com/ArmanJR/termustat/blob/0d05ced37a29244cff7652025375ae34bc954345/api/repositories/course.go

9 10 13 37 43 **github.com**
https://github.com/ArmanJR/termustat/blob/0d05ced37a29244cff7652025375ae34bc954345/api/handlers/auth.go

11 **github.com**
https://github.com/ArmanJR/termustat/blob/0d05ced37a29244cff7652025375ae34bc954345/api/errors/errors.go

12 16 17 32 39 41 42 **github.com**
https://github.com/ArmanJR/termustat/blob/0d05ced37a29244cff7652025375ae34bc954345/api/services/auth.go

15 49 **github.com**
https://github.com/ArmanJR/termustat/blob/0d05ced37a29244cff7652025375ae34bc954345/api/services/faculty.go

18 40 **github.com**
https://github.com/ArmanJR/termustat/blob/0d05ced37a29244cff7652025375ae34bc954345/api/database/migrations/000001_initial.up.sql

19 20 23 24 26 74 78 **github.com**
https://github.com/ArmanJR/termustat/blob/0d05ced37a29244cff7652025375ae34bc954345/api/routes/routes.go

21 22 36 48 54 65 **github.com**
https://github.com/ArmanJR/termustat/blob/0d05ced37a29244cff7652025375ae34bc954345/api/handlers/course.go

25 **github.com**
https://github.com/ArmanJR/termustat/blob/0d05ced37a29244cff7652025375ae34bc954345/api/handlers/faculty.go

27 28 29 30 **github.com**
https://github.com/ArmanJR/termustat/blob/0d05ced37a29244cff7652025375ae34bc954345/api/middlewares/jwt.go

31 **github.com**
https://github.com/ArmanJR/termustat/blob/0d05ced37a29244cff7652025375ae34bc954345/api/middlewares/is_admin.go

33 34 66 75 **github.com**
https://github.com/ArmanJR/termustat/blob/0d05ced37a29244cff7652025375ae34bc954345/api/dto/course.go

35 38 45 46 53 64 **github.com**
https://github.com/ArmanJR/termustat/blob/0d05ced37a29244cff7652025375ae34bc954345/api/handlers/university.go

44 **github.com**
https://github.com/ArmanJR/termustat/blob/0d05ced37a29244cff7652025375ae34bc954345/api/main.go

50 51 52 76 **github.com**
https://github.com/ArmanJR/termustat/blob/0d05ced37a29244cff7652025375ae34bc954345/readme.md

55 56 **github.com**
https://github.com/ArmanJR/termustat/blob/0d05ced37a29244cff7652025375ae34bc954345/api/services/university.go

61 67 77 **github.com**
https://github.com/ArmanJR/termustat/blob/0d05ced37a29244cff7652025375ae34bc954345/api/services/user_course.go

68 **github.com**
https://github.com/ArmanJR/termustat/blob/0d05ced37a29244cff7652025375ae34bc954345/api/database/database.go