ChatGPT

Durability AI System Blueprint

Core Entities and Schema

This section describes the schema for all core entities in the Durability AI system. Each entity is represented as a table in the database, with key fields and relationships outlined below.

Pattern Types

Definition: Categories that classify patterns by training modality or focus (e.g., Strength vs Endurance).

- **id** (PK): Unique identifier for the pattern type (integer primary key).
- **name**: Text name of the pattern type (e.g., "Strength", "Endurance"). *Not null*.
- **description**: Text description of this category.

Example: Pattern types might include **Strength**, **Endurance**, **Mobility**, **Stability**, etc., to label the nature of patterns and exercises.

Patterns

Definition: Fundamental movement or training patterns. Each pattern belongs to a pattern type.

- **id** (PK): Unique identifier for the pattern.
- **name**: Name of the movement pattern (e.g., "Squat", "Hinge", "Vertical Pull"). *Not null*.
- **pattern_type_id** (FK): Reference to the **Pattern Types** table, indicating the category of this pattern.
- **description**: Text description of the pattern.

Each pattern corresponds to a fundamental movement or exercise category (for example, *Squat*, *Hinge*, *Lunge*, *Push*, *Pull*, *Carry*, etc.). The `pattern_type_id` associates it with a modality like Strength or Stability. For instance, "Squat" may be categorized under the Strength pattern type.

Exercises

Definition: Specific exercises or movements. Each exercise is linked to a pattern.

- **id** (PK): Unique identifier for the exercise.
- **name**: Name of the exercise (e.g., "Barbell Back Squat", "Plank"). *Not null*.
- **pattern_id** (FK): Reference to the **Patterns** table (the movement pattern this exercise exemplifies).
- **description**: Text description of the exercise (what it is and how it is performed).
- **level**: Difficulty or proficiency level (e.g., Beginner, Intermediate, Advanced).

Exercises are the concrete activities performed by users. They inherit classification from their associated pattern (and pattern type). The `level` field provides a general indication of difficulty or progression stage.

Instead of a fixed field for level, difficulty can also be represented via tags (see **Tags/Attributes** below), which provides more flexibility. Each exercise can be associated with multiple tags for richer classification.

Tags / Attributes

Definition: Flexible labels to tag exercises (or patterns) with additional attributes such as required equipment, target muscle group, difficulty, etc.

- **id** (PK): Unique identifier for the tag/attribute.
- **name**: Text of the tag (e.g., "Bodyweight", "Barbell", "Cardio", "Beginner"). *Not null*.
- **category**: Category of the tag to group similar attributes (e.g., Equipment, Difficulty, Modality, BodyPart).

Tags allow filtering and extended metadata. For example, an exercise may have tags for **equipment** (Bodyweight, Dumbbell, Machine), **difficulty level** (Beginner/Intermediate/Advanced), or other attributes like **Cardio**, **Strength**, **Mobility**, **Balance**. Tags are linked to exercises through an association table (many-to-many relationship), since an exercise can have multiple tags.

Note: In the database, a join table **exercise_tags** is used to associate exercises with tags. It contains `exercise_id` and `tag_id` pairs for each assignment. This allows queries like "find all Beginner exercises requiring no equipment."

Metrics

Definition: Atomic performance metrics that can be recorded or used to quantify exercises.

- **id** (PK): Unique identifier for the metric.
- **name**: Name of the metric (e.g., "Weight", "Repetitions", "Duration"). *Not null*.
- **unit**: Unit of measurement (e.g., "kg", "count", "seconds", "bpm").
- **description**: Description of the metric's meaning or usage.

Metrics represent raw data points (inputs or outputs) for exercise performance. For instance, a Strength exercise might use **Weight** (kg) and **Repetitions**, while an Endurance exercise might use **Duration** or **Heart Rate**. These metrics feed into calculations for intensity and scoring.

Supermetrics

Definition: Composite or aggregated metrics, often derived from basic metrics, to track higher-level performance or scoring.

- **id** (PK): Unique identifier for the supermetric.
- **name**: Name of the supermetric (e.g., "Total Volume", "Durability Score").
- **description**: Explanation of what the supermetric represents or how it's calculated.

Supermetrics combine multiple data points or metrics. For example, **Total Volume** could be computed as `Weight * Reps` summed across sets (a measure of work done in a strength exercise). A **Durability Score** might aggregate strength, stability, and mobility metrics to give an overall indicator of durability. These

computations are defined in code or derived via queries rather than being directly stored (though they could be materialized for performance).

Phases

Definition: Training phases or program stages that an exercise or pattern might be associated with.

- **id** (PK): Unique identifier for the phase.
- **name**: Name of the phase (e.g., "Phase 1 - Recovery", "Phase 3 - Strength").
- **description**: Description of the training phase's focus.

Phases can be used to periodize or categorize programs (for example, **Recovery**, **Foundation**, **Strength**, **Performance** phases). In practice, one might tag certain exercises or patterns as appropriate for certain phases (e.g., basic mobility drills in Phase 1, heavy lifts in Phase 3). While this blueprint doesn't create an explicit link table between exercises and phases, such relationships can be added (e.g., a table linking exercises to phases if needed for program generation).

Injuries

Definition: Injuries or conditions that need to be considered when recommending exercises (for contraindications or rehabilitation).

- **id** (PK): Unique identifier for the injury/condition.
- **name**: Name of the injury or condition (e.g., "Knee Pain", "Shoulder Impingement"). *Not null*.
- **description**: Description of the injury, context, or severity notes.

This table enumerates common injuries or physical conditions that affect exercise selection. The injury data is used to filter out contraindicated exercises and suggest indicated ones (rehabilitative or supportive exercises).

Exercise Contraindications / Indications: There are two association tables to map relationships between **Exercises** and **Injuries**: - **exercise_contraindications**: Lists pairs of (exercise_id, injury_id) where performing that exercise could aggravate the injury (i.e., the exercise is *contraindicated* for that injury). - **exercise_indications**: Lists pairs of (exercise_id, injury_id) where the exercise is beneficial or recommended for the injury (i.e., an *indicated* rehab or prehab exercise).

These mappings are used in logic to exclude harmful exercises and include helpful ones based on a user's injury profile.

Context

Definition: Contextual scenarios or conditions under which a program is generated. Context influences exercise selection and parameters.

- **id** (PK): Unique identifier for the context.
- **name**: Name of the context scenario (e.g., "Home / No Equipment", "Rehab / Low Intensity"). *Not null*.

- **description**: Details about the context (equipment availability, time constraints, goals).
- **equipment_available**: Boolean indicating if substantial equipment is available (`TRUE` for full gym equipment, `FALSE` for bodyweight/minimal equipment scenarios).
- **intensity_level**: Desired intensity level for the session (e.g., "Low", "Medium", "High").
- **time_limit**: Time constraint in minutes (if applicable; e.g., 15 for a quick workout, or NULL for none).

Contexts encapsulate external factors and user goals. For example, a **Home / No Equipment** context will restrict exercises to those with no equipment requirement, and likely moderate intensity. A **Rehab** context implies low intensity and a focus on safe movements. Context definitions can be used by filtering logic to tailor exercise selection (see filtering pseudocode below). In the database, context can be a standalone table; if more complex rules are needed, context can also be linked to specific tag requirements (e.g., context "No Equipment" could be associated with a rule that equipment tags must be "Bodyweight").

Scoring and Intensity Normalization

To compare effort across different types of exercises, Durability AI uses **intensity normalization policies**. These policies define how raw metrics translate into a unified intensity score or category. Each pattern type can have its own scoring logic. Below is a summary of how intensity is handled for each major category (pattern type):

Pattern Type	Primary Metrics	Normalization Policy
Strength	Weight, Reps (and optionally 1RM)	Intensity is calculated relative to the individual's capacity. For example, weight can be expressed as a percentage of one-rep max. A heavy set (near 100% of 1RM for few reps) would yield a high intensity score (close to 100). More reps at lower weight produce a moderate intensity. RPE (Rate of Perceived Exertion) can also be used to fine-tune the score. A policy table or formula (e.g., using the Epley formula or an RPE conversion chart) can map weight and reps to a normalized 0–10 or 0–100 intensity value.
Endurance	Heart Rate, Pace, Duration	Intensity is based on cardiovascular effort. For example, heart rate as a percentage of max heart rate or pace as a fraction of personal best. Running or cycling at max sustainable speed would be high intensity. The normalization might map heart rate zones to intensity scores (e.g., Zone 2 = low, Zone 5 = very high).
Mobility	Range of motion, Discomfort level	Intensity for mobility exercises is qualitative. A possible approach is to score intensity by the difficulty or discomfort of the stretch. If a stretch is taken to near-end range (significant effort or discomfort), label it as higher intensity. Otherwise gentle mobility work scores low. This may be simplified to categorical values (e.g., light, moderate stretch) since quantitative metrics are harder.

Pattern Type	Primary Metrics	Normalization Policy
Stability	Hold time, Stability challenge	Intensity for stability/balance exercises might be derived from how challenging the position or movement is and how long it can be held. For instance, a basic plank might be medium intensity if held to fatigue, whereas an advanced balance exercise on an unstable surface could rate high. Normalization might use time (longer hold = higher endurance but possibly lower intensity per second) combined with a difficulty factor. Often a simple Low/Medium/High classification is used for stability challenges.

The system may use lookup tables or formulas to implement these policies. For example, a **Strength Policy** table could define intensity multipliers for rep counts at a given %1RM, ensuring consistent scoring. Similarly, a **Heart Rate Zone** table could map beats-per-minute ranges to a 1–10 intensity scale. All exercises then get an intensity score which can be used to balance workouts or ensure a target intensity range is met.

Normalization Example: If a user performs a squat with 80 kg for 5 reps and their estimated 1RM is 100 kg, that's 80% of 1RM for 5 reps, which might correspond to ~8/10 intensity (vigorous but not maximal). A 30-minute jog at 60% max heart rate might be considered ~5/10 intensity (moderate). These scores allow the system to compare or limit workloads across exercise types.

Versioning and Deployment Guidance

Maintaining the system blueprint requires careful versioning as the schema and data evolve:

- **Schema Versioning:** Assign a version number to the database schema (and possibly seed data). For example, use semantic versioning (v1.0, v1.1, etc.) whenever you add or modify tables and fields. Keep a changelog in the documentation or in the repository (e.g., a `CHANGELOG.md` that notes new columns or entities).
- **Migrations:** Use Supabase CLI or a migration tool to apply schema changes. Each change (adding a table, altering a column) should be done via a SQL migration script. This ensures reproducibility. For instance, use `supabase migration create "add-new-entity"` to generate a migration file, then fill in the `CREATE TABLE` or `ALTER TABLE` statements.
- **Seed Data Versioning:** Treat the seed data as part of the version. If core seed data (like a new pattern or exercise) is added or changed, update the seed files and bump a minor version. Provide a clear path for deploying these changes (for example, a fresh seed insert or an UPSERT logic).
- **Deployment Workflow:** In a team setting, maintain the schema in source control (SQL files or a declarative format). On deployment, apply migrations in order, then load or sync seed data. Supabase CLI can run `supabase db push` to apply schema to the linked database. Alternatively, use `pgdump` for a baseline schema and separate seed import scripts for data.
- **Backward Compatibility:** If releasing a new version of the schema, consider writing migrations to transform existing production data to fit the new schema (for example, populating a new column with default values or splitting a field).
- **Testing Upgrades:** Try the migration on a staging database with a copy of data to ensure it doesn't break existing queries or logic. Only then apply to production.

By following versioning best practices, the Durability AI data model can be safely updated over time without disrupting service or losing data.

Security and RLS Guidance (Supabase)

Security is critical since Durability AI stores user-specific data (e.g., personal workout info, possibly health info). Supabase uses PostgreSQL Row Level Security (RLS) to enforce access control. Key guidelines:

- **Enable RLS on all tables containing user data:** Supabase recommends enabling RLS on any table in the exposed schema (typically `public`) [1]. When RLS is enabled, by default no rows are returned unless policies permit it.
- **Default to least privilege:** After enabling RLS, define explicit policies for each table:
- For reference tables that are non-sensitive (like Patterns, Exercises, Tags), you can allow read access to all users while restricting writes. For example, allow `SELECT` for role `authenticated` (and `anon` if you want even non-logged-in users to read) with condition `TRUE` (no restriction), but no `INSERT/UPDATE` except for admins or service role.
- For user-specific tables (e.g., if there was a `user_progress` table or personal notes), create policies tying records to `auth.uid()`. For instance, a policy to allow users to see only their own records would use something like: `CREATE POLICY ... USING ((SELECT auth.uid()) = user_id)` [2].
- **Use Supabase Auth roles:** Supabase provides an `authenticated` role for logged-in users and `anon` for others. Use the `TO authenticated` or `TO anon` clause in policies to differentiate open access from restricted. For example, you might allow `authenticated` users to `SELECT` from the Exercises table but no one to modify it, and no access at all for `anon` if your API should only serve logged-in users.
- **Service Role for privileged actions:** In Supabase, the service key (used on the backend) can bypass RLS. Design your backend functions or admin interface to use the service role when seeding or modifying core data. Regular clients (using JWT auth) should operate under RLS restrictions.
- **Row Level Security Policies:** Write policies for each table according to use case:
- **Core lookup tables (Patterns, Exercises, etc.):** Enable RLS, then allow full read (`SELECT`) to all authenticated users [3]. Disallow `INSERT/UPDATE/DELETE` for regular users; only allow these via service role or a role check (e.g., if you had an `is_admin` claim).
- **User-specific tables (if any):** E.g., a hypothetical `user_workout_log` table would have `ENABLE RLS` and a policy `USING (user_id = auth.uid())` to ensure each user only sees their rows. You would also add a `WITH CHECK (user_id = auth.uid())` on `INSERT/UPDATE` to prevent one user spoofing another's ID.
- **Verify policy effects:** Remember that once RLS is enabled on a table, any query with the anon (public) role will return nothing unless a policy allows it [4]. Always test as an unauthenticated and as an authenticated user to make sure the policies are correctly restricting access.
- **Encryption & sensitive data:** If any personal health data or identifiable information is stored, consider using additional encryption (Supabase offers extensions like `pgcrypto`). Also, apply the principle of minimum data retrieval – queries should only select the necessary fields (which is more of an application concern, but worth noting).
- **Monitoring and Logging:** Use Supabase's logging to monitor if any policy rejections occur (for example, if a client tries to access something they shouldn't, the 403 errors can be logged). This can help fine-tune policies.

In summary, **enable RLS on all tables in the public schema and create strict policies** [1] [4]. The combination of RLS and Supabase Auth ensures that each user or role only accesses allowed data. This provides end-to-end security from the database level upward.

Testing and Validation Procedures

Ensuring that the blueprint is correctly implemented requires tests at multiple levels:

- **Schema Validation:** Verify that all foreign key relationships are intact and working. For example, inserting an Exercise with a non-existent `pattern_id` should fail. Ensure that NOT NULL constraints are present where expected (names, required fields). One can write basic tests or use a tool (like Postgres' `pgTAP` extension) to assert schema properties.
- **Seed Data Checks:** After loading seed data (the CSV/JSON files provided), run sanity queries. For instance:
- Count that each patterns' `pattern_type_id` actually exists in Pattern Types.
- Pick a sample exercise and join through pattern to pattern_type to confirm the relationships.
- Ensure no duplicate tags or other unique fields.
- **Business Logic Tests:** Use the pseudocode logic on known scenarios to see if results match expectations:
- *Intensity Calculation:* Take a known exercise scenario (like a weight and rep for strength, or a duration for cardio) and compute intensity. Check that it falls in the expected range or category. For example, if a user performs a very easy set, verify the intensity comes out low.
- *Filtering by Context/Tags:* Simulate a context and verify filtering. For a "Home / No Equipment" context, ensure that none of the exercises selected require equipment (i.e., all have the "Bodyweight" tag or similar). If the context has `intensity_level = Low`, ensure high-intensity exercises (like very heavy lifts) are filtered out or at least flagged.
- *Contraindication Logic:* Create a test user profile with an injury (e.g., Knee Pain) and a context, run the exercise selection routine, and confirm that no contraindicated exercises for Knee Pain are present. Additionally, verify that indicated exercises for Knee Pain are included or prioritized. For example, if "Knee Pain" is active, ensure squats or lunges above a certain intensity are removed, but rehab moves like TKE or Clamshell are present.
- **Unit Testing Functions:** If you implement the intensity calculation or filtering as database functions or in application code, write unit tests for those functions. For instance, a test for `calculate_intensity(strengthExerciseInput)` should assert known outputs for given inputs (maybe using fixed 1RM assumptions).
- **Integration Testing:** Use a staging environment of the Supabase DB with the schema and run end-to-end scenarios:
- Insert a fake user and assign them a context and an injury.
- Run the stored procedure or API that generates a workout prescription.
- Check that the returned exercises respect all rules (context filters, no contraindications, correct intensities).
- Also ensure RLS policies don't inadvertently block the procedure or the data retrieval for that user.
- **RLS Policy Testing:** As mentioned, test with different roles. In Supabase, you can use the JWT of a test user to simulate authenticated requests and also call the API with no JWT for anon. Verify:
- Authenticated user can read the reference tables (Patterns, Exercises).
- Authenticated user cannot write to them (try an insert via the API, expect rejection).
- Authenticated user can insert and read their own user-specific records (if such tables exist).

- Anonymous cannot read data that should require auth.
- The service role (using the secret key) can perform admin operations (like seeding or bulk updates).
- **Performance Validation:** Though not strictly asked, it's wise to ensure that the filtering logic and intensity calculations perform under load. For instance, if the recommendation query joins many tables (exercises, tags, injuries), ensure indexes are in place (e.g., on `exercise_tags(exercise_id)` and `exercise_tags(tag_id)`, etc.). Write tests or use EXPLAIN plans to confirm that queries use indexes.

By following these testing steps, you validate that the system blueprint works as intended and can be confidently deployed. Testing should be part of the development cycle whenever the schema or logic is updated (for example, re-run these tests after any version upgrade as mentioned in **Versioning**).

Pseudocode for Key Logic

To implement the core logic of Durability AI, we outline pseudocode for the primary decision-making components: intensity calculation, context/tag filtering, and contraindication checks. This pseudocode can be used to guide actual code or database function implementations.

Intensity Calculation by Pattern Type

This function computes a normalized intensity value for a given exercise attempt, based on the exercise's pattern type and relevant metrics.

```
function calculate_intensity(exercise, metrics):
    # Determine pattern type of the exercise (e.g., Strength, Endurance, etc.)
    type = exercise.pattern.pattern_type.name

    if type == "Strength":
        # Assume metrics include weight (kg) and reps
        weight = metrics["weight"]   # weight used in kg
        reps = metrics["reps"]       # repetitions performed
        # If 1RM (one-rep max) known (could be precomputed or stored per user):
        if metrics.contains("one_rm"):
            one_rm = metrics["one_rm"]
        else:
            # Estimate 1RM from weight and reps (e.g., Epley formula)
            one_rm = weight * (1 + reps / 30.0)
        intensity_percent = (weight / one_rm) * 100    # percentage of max
        # Adjust by reps: if doing high reps at lower %1RM, intensity might
still be moderate
        # For simplicity, you might take intensity_percent as is, or use a
table.
        intensity = intensity_percent                # output on 0-100 scale
        if intensity > 100: intensity = 100  # cap at 100%

    elseif type == "Endurance":
```

```
        # Use heart rate or pace to determine intensity
        hr = metrics.get("heart_rate")        # current heart rate
        max_hr = metrics.get("max_heart_rate")  # user max heart rate
        if hr and max_hr:
            intensity = (hr / max_hr) * 100   # percentage of max HR
        else:
            # Alternatively, use perceived exertion or pace
            rpe = metrics.get("RPE")  # 1-10 scale
            if rpe:
                intensity = (rpe / 10.0) * 100  # convert RPE to %
            else:
                intensity = 0  # default if no data
        # Cap and categorize if needed
        if intensity > 100: intensity = 100

    elseif type == "Mobility":
        # Mobility intensity might use subjective measures
        discomfort = metrics.get("discomfort_level")  # e.g., 1-10
        if discomfort:
            intensity = (discomfort / 10.0) * 100
        else:
            # If stretch was easy and full range, call it low intensity
            intensity = 20  # default low intensity
        # Mobility generally kept low/moderate to avoid injury

    elseif type == "Stability":
        # Stability intensity could be based on how challenging the variation is
        difficulty = metrics.get("difficulty")  # e.g., a score or level of the
exercise variation
        hold_time = metrics.get("hold_time")    # how long (seconds) held, if
relevant
        if difficulty:
            intensity = difficulty * 20  # assuming difficulty is 1-5 scale,
convert to % (this is arbitrary)
        else:
            intensity = 0
        # Optionally adjust intensity by hold time (longer hold might increase
endurance demand but not peak intensity)
        # For simplicity, ignore hold_time or use it to slightly adjust
intensity.

    else:
        # Unknown type, default
        intensity = metrics.get("RPE", 50)  # use RPE if available, else 50 as
medium

    # Final normalization
    if intensity < 0: intensity = 0
```

```
    if intensity > 100: intensity = 100
    return intensity  # as a percentage (0-100) or could convert to 0-10 scale
if needed
```

Explanation: This pseudocode distinguishes intensity calculation by pattern type. Strength uses weight and reps (and ideally an estimated 1RM) to gauge intensity. Endurance uses heart rate or an RPE if HR is not available. Mobility and Stability rely on more qualitative metrics since they're harder to quantify (we used placeholders like `discomfort_level` or a `difficulty` rating). In practice, these would be refined with real data or expert input. The function returns a normalized percentage that can be used consistently across exercise types.

Filtering Logic by Context/Tags

This function filters a list of candidate exercises based on a given context and optional required tags. It ensures that the returned exercises fit the scenario.

```
function filter_exercises(exercises, context, required_tags=[]):
    filtered = []

    for exercise in exercises:
        # Check equipment availability
        if context.equipment_available == false:
            # Context has no equipment: exclude exercises needing equipment
            if exercise has any tag in ["Barbell", "Dumbbell", "Machine",
"Kettlebell"]:
                continue  # skip this exercise (requires equipment not
available)
        # If context allows equipment, we don't filter by equipment tags (all
good).

        # Check intensity level filtering
        if context.intensity_level:
            # For simplicity, define a mapping of intensity_level to threshold
            level = context.intensity_level  # e.g., "Low", "Medium", "High"
            exercise_intensity = estimate_exercise_intensity(exercise)  # could
use pattern type defaults or metadata
            if level == "Low" and exercise_intensity > 33:
                continue  # skip exercises too intense for a low-intensity
session
            if level == "Medium" and exercise_intensity > 66:
                continue  # skip very intense exercises in a moderate session
            # (If level == "High", we generally allow all, or could ensure some
minimum intensity)

        # Check required tags
        allow = true
```

```
        for tag in required_tags:
            if not exercise has tag:
                allow = false
                break
        if not allow:
            continue  # this exercise doesn't have a tag that the context or
user specifically requires

        # If we reach here, exercise passed all filters
        filtered.append(exercise)

    return filtered
```

Explanation: The filtering logic goes through each exercise and applies rules: - Equipment: If the context indicates no equipment, any exercise tagged with equipment (barbell, machine, etc.) is excluded. - Intensity: We compare the context's desired intensity level with an estimated intensity of the exercise. If the exercise is inherently too intense for a low or medium session, it's skipped. (For example, heavy deadlifts might be filtered out of a "Low" intensity context). - Required Tags: If the user or context specifically needs certain attributes (like an exercise that targets "Mobility" or a certain muscle group), we ensure each exercise has those tags. Only exercises containing all required tags are kept.

The `estimate_exercise_intensity(exercise)` in pseudocode stands in for a function that could use exercise metadata (perhaps the pattern type or known difficulty) to judge how intense that exercise typically is. In implementation, we might store a default intensity or difficulty rating with each exercise to aid this filter (e.g., Sprinting is high, Walking is low).

The result is a list of exercises that fit the context's constraints. This would be used as a pool to then apply the injury contraindication filter.

Contraindication/Indication Logic Based on Injury

This logic takes a user's injury profile into account when finalizing exercise selection. It removes contraindicated exercises and can optionally boost or include indicated ones.

```
function apply_injury_filters(exercises, user_injuries):
    safe_exercises = []

    for exercise in exercises:
        # Assume user_injuries is a list of injury IDs (or injury objects).
        contraindicated = false

        for inj in user_injuries:
            # Check if this exercise is contraindicated for this injury
            if (exercise.id, inj.id) is in exercise_contraindications_table:
                contraindicated = true
                break
```

```
        if contraindicated:
            continue  # skip this exercise entirely

        # At this point, exercise is not contraindicated for any user injury.
        # (Optional) Check if it's indicated for any injury:
        exercise_score = 0
        for inj in user_injuries:
            if (exercise.id, inj.id) is in exercise_indications_table:
                # If it's a specifically recommended exercise for the injury,
mark it.
                exercise_score += 1  # simple way to prioritize indicated
exercises
        exercise.indication_score = exercise_score

        safe_exercises.append(exercise)

    # After filtering, you might sort or prioritize indicated exercises
    safe_exercises.sort(by="indication_score", descending=true)

    return safe_exercises
```

Explanation: This function iterates over the candidate exercises (already filtered by context perhaps) and excludes any that appear in the **exercise_contraindications** mapping for any of the user's injuries. Those that pass are considered "safe." We then optionally check against **exercise_indications**; if an exercise is known to be beneficial for a condition the user has, we can mark or score it. In this pseudocode, we give it a simple `indication_score` (e.g., 1 point per relevant injury). Finally, one could sort exercises so that indicated ones are ranked higher or ensure at least one indicated exercise is included in a prescription.

In practice, this could be done in a database query (joining exercises with the injury tables and filtering), or in application logic after querying a list of possible exercises. The outcome is that a user with injuries will not get harmful exercises and is more likely to get helpful ones. For example, a user with "Knee Pain" might have heavy squats and running filtered out, while clamshells and TKEs (Terminal Knee Extensions) are flagged as good and thus chosen.

SQL Schema Definitions

Below are the SQL `CREATE TABLE` statements for each core entity and the relevant association tables, followed by some sample `INSERT` statements using the seed data. These definitions align with the schema described above.

```sql
-- Pattern Types table
CREATE TABLE pattern_types (
    id SERIAL PRIMARY KEY,
    name TEXT NOT NULL UNIQUE,
    description TEXT
);
```

```sql
-- Patterns table (with FK to Pattern Types)
CREATE TABLE patterns (
    id SERIAL PRIMARY KEY,
    name TEXT NOT NULL,
    pattern_type_id INT NOT NULL REFERENCES pattern_types(id),
    description TEXT
);

-- Exercises table (with FK to Patterns)
CREATE TABLE exercises (
    id SERIAL PRIMARY KEY,
    name TEXT NOT NULL,
    pattern_id INT NOT NULL REFERENCES patterns(id),
    description TEXT,
    level TEXT
    -- 'level' could alternatively be normalized into tags or a separate table
);

-- Tags table
CREATE TABLE tags (
    id SERIAL PRIMARY KEY,
    name TEXT NOT NULL UNIQUE,
    category TEXT
);

-- Contexts table
CREATE TABLE contexts (
    id SERIAL PRIMARY KEY,
    name TEXT NOT NULL,
    description TEXT,
    equipment_available BOOLEAN NOT NULL DEFAULT FALSE,
    intensity_level TEXT,
    time_limit INT
);

-- Metrics table
CREATE TABLE metrics (
    id SERIAL PRIMARY KEY,
    name TEXT NOT NULL UNIQUE,
    unit TEXT,
    description TEXT
);

-- Supermetrics table
CREATE TABLE supermetrics (
    id SERIAL PRIMARY KEY,
    name TEXT NOT NULL UNIQUE,
```

```
    description TEXT
);

-- Phases table
CREATE TABLE phases (
    id SERIAL PRIMARY KEY,
    name TEXT NOT NULL UNIQUE,
    description TEXT
);

-- Injuries table
CREATE TABLE injuries (
    id SERIAL PRIMARY KEY,
    name TEXT NOT NULL UNIQUE,
    description TEXT
);

-- Association table for Exercise-Tag (many-to-many)
CREATE TABLE exercise_tags (
    exercise_id INT REFERENCES exercises(id),
    tag_id INT REFERENCES tags(id),
    PRIMARY KEY (exercise_id, tag_id)
);

-- Association table for contraindications (Exercise-Injury)
CREATE TABLE exercise_contraindications (
    exercise_id INT REFERENCES exercises(id),
    injury_id INT REFERENCES injuries(id),
    PRIMARY KEY (exercise_id, injury_id)
);

-- Association table for indications (Exercise-Injury)
CREATE TABLE exercise_indications (
    exercise_id INT REFERENCES exercises(id),
    injury_id INT REFERENCES injuries(id),
    PRIMARY KEY (exercise_id, injury_id)
);
```

The SQL above defines primary keys, foreign keys, and basic constraints (like some `NOT NULL` and `UNIQUE` where appropriate). In a real setup, you might add additional constraints (for example, ensure `intensity_level` in contexts is one of a set of allowed values, via a CHECK constraint or use of an enum type).

Sample Inserts: Below are a few examples of inserting seed data into these tables. (The full seed data is provided in the accompanying CSV/JSON files.)

```sql
-- Insert some pattern types
INSERT INTO pattern_types (name, description) VALUES
('Strength', 'Strength-focused patterns (resistance training)'),
('Endurance', 'Endurance-focused patterns (cardio, conditioning)'),
('Mobility', 'Mobility-focused patterns (flexibility, range of motion)'),
('Stability', 'Stability-focused patterns (balance, core stability)');

-- Insert a couple of patterns
INSERT INTO patterns (name, pattern_type_id, description) VALUES
('Squat', 1, 'Knee-dominant lower body movement'),
('Vertical Pull', 1, 'Upper body vertical pulling movement (e.g., pull-ups)'),
('Rotation', 4, 'Core rotational movement pattern');

-- Insert example exercises
INSERT INTO exercises (name, pattern_id, description, level) VALUES
('Barbell Back Squat', 1, 'Barbell squat with weight on back', 'Advanced'),
('Pull-Up', 2, 'Bodyweight pull-up exercise', 'Advanced'),
('Russian Twist', 3, 'Seated core rotation exercise (Russian Twist)',
'Beginner');

-- Insert tags and exercise-tag relations
INSERT INTO tags (name, category) VALUES
('Bodyweight', 'Equipment'),
('Barbell', 'Equipment'),
('Beginner', 'Difficulty');
-- Link an exercise to tags (e.g., mark Pull-Up as Bodyweight and Advanced)
INSERT INTO exercise_tags (exercise_id, tag_id) VALUES
(2, 1),  -- Pull-Up is a Bodyweight exercise
(1, 2);  -- Barbell Back Squat uses Barbell

-- Insert injuries and contraindication mappings
INSERT INTO injuries (name, description) VALUES
('Knee Pain', 'Generic knee pain condition'),
('Shoulder Impingement', 'Shoulder impingement injury');
-- Map a contraindication: Barbell Back Squat (id=1) is contraindicated for Knee
Pain (id=1)
INSERT INTO exercise_contraindications (exercise_id, injury_id) VALUES (1, 1);
```

These SQL statements illustrate how the schema is implemented and how seed data populates the tables. In a production setup, you would likely execute a series of such insert statements (or use a bulk import for large datasets) to seed the database with all patterns, exercises, tags, etc., from the CSV/JSON files.

All the above files (the markdown documentation, seed data CSV/JSON files, and an SQL schema file) are packaged in the attached ZIP archive for convenience. This structure can be loaded into Supabase (using the CLI or Studio) or into any Postgres instance to bootstrap the Durability AI database.

Package Contents (DurabilityAI_Package.zip):

```
blueprint.md             -- Comprehensive system blueprint (this document)
schema.sql               -- SQL schema definitions for all tables
pattern_types.csv        -- Seed data for Pattern Types
patterns.csv             -- Seed data for Patterns
exercises.csv            -- Seed data for Exercises
tags.csv                 -- Seed data for Tags/Attributes
contexts.csv             -- Seed data for Context scenarios
metrics.csv              -- Seed data for Metrics
supermetrics.csv         -- Seed data for Supermetrics
phases.csv               -- Seed data for Phases
injuries.csv             -- Seed data for Injuries
exercise_tags.csv        -- Seed mapping of Exercises to Tags
exercise_contraindications.csv -- Seed mapping of contraindicated Exercise-
Injury pairs
exercise_indications.csv -- Seed mapping of indicated Exercise-Injury pairs
home_no_equipment.json   -- Example exercise prescription for a Home/No-
Equipment context
gym_full_equipment.json  -- Example exercise prescription for a Gym/Full-
Equipment context
rehab_knee_pain.json     -- Example exercise prescription for a Rehab (Knee
Pain) context
```

This structure is ready to use with the Supabase CLI or in the Cursor development environment. You can load the schema (via `schema.sql`), then import the CSV data into the respective tables, and run queries or application code using the provided JSON examples as references.

---

[1] [2] [3] [4] Row Level Security | Supabase Docs
https://supabase.com/docs/guides/database/postgres/row-level-security