ChatGPT

Backend Technical Documentation: Shopify Product Upload Workflow Application

Overview: This document describes the back-end architecture and implementation of a standalone web application that manages **product upload workflows** to a Shopify store. The application is built with **Node.js (Express)** and **PostgreSQL**, and is deployed on **Replit**. It covers the module structure (with each module's purpose and responsibilities after refactoring), full API reference (endpoints, examples, status codes, authentication, and permissions), authentication and authorization (JWT-based login and role-based access control), database schema and migrations, deployment steps on Replit, and the internal **state machine** logic with checklist enforcement for product workflow transitions. This documentation is intended for developers maintaining or expanding the backend.

Module Structure and Responsibilities

The codebase is organized into clear modules, each handling a specific concern (following separation of concerns). After refactoring, each module has well-defined responsibilities, which improves maintainability and scalability. Below is an outline of the main modules and their roles:

Express App & Server Module

- **Components:** `app.js` (or `server.js`) entry point, Express app configuration, and main server startup code.
- **Purpose:** Initializes the Express application, sets up global middleware, loads route modules, and starts the HTTP server. It may also handle global error handling.
- **Details:** This module imports all other modules (routes/controllers) and applies middleware like JSON body parser (`express.json()`), CORS if needed, etc. It typically listens on `process.env.PORT` (using `0.0.0.0` host for Replit). For example, it might look like:

```
const app = express();
app.use(express.json());
// ... possibly app.use(cors()) or other middleware
app.use('/api/auth', authRoutes);
app.use('/api/products', productRoutes);
// ... other route mounts
app.use(errorHandler); // global error handling middleware
app.listen(process.env.PORT || 3000, '0.0.0.0');
```

The server module ensures all routes are registered and uses an `errorHandler` middleware (if implemented) to catch any unhandled errors and respond with a 500 status.

Authentication & Authorization Module

- **Components:** `authController.js`, `authService.js`, `authMiddleware.js`, and `roleMiddleware.js` (or similar naming).
- **Purpose:** Manages user authentication (login) and enforces authorization on protected routes using JWTs and roles.
- **Responsibilities:**
- The **Auth Controller** handles the login endpoint. It validates user credentials and returns a signed JWT on successful login.
- The **Auth Service** encapsulates business logic for authentication (e.g., verifying password hash with the database, generating JWT tokens using a secret key).
- Passwords are stored securely: when a new user is created (e.g., seeding an admin), the password is hashed (using a strong one-way hash like bcrypt) and only the hash is stored in PostgreSQL. On login, the provided password is compared with the stored hash.
- The **Auth Middleware** (`authMiddleware.js`) is an Express middleware that checks for a JWT in the `Authorization` header on incoming requests. It verifies the token's signature (using the secret key from environment variables) and, if valid, attaches the decoded user info (e.g. `{ id, username, role }`) to `req.user`. If the token is missing or invalid, it returns a 401 Unauthorized response.
- The **Role Middleware** (e.g., `requireRole`) enforces role-based access control. For example, `requireRole('admin')` will only call `next()` if `req.user.role` is `'admin'`, otherwise respond with 403 Forbidden. This can also be extended to accept an array of roles for endpoints accessible by multiple roles.
- **Refactored Approach:** After refactoring, all authentication logic is encapsulated here (the controllers are kept thin, simply calling `AuthService` for verification and token creation). Authorization checks are handled consistently via middleware instead of scattered in controllers. This module ensures that protected routes cannot be accessed without a valid token and appropriate user role.

User Management Module (User Model)

- **Components:** `userModel.js` (or part of an ORM models definition) and possibly a `userController.js` for any user-related endpoints.
- **Purpose:** Defines the user data model and provides an interface to the users table in the database. Also may handle any endpoints related to user management (though in this application, user management is minimal).
- **Details:** The **User Model** defines the schema for users (fields like id, username, password_hash, role, etc.) and provides functions to query users (e.g., find by username). In a simple setup, this could just be raw SQL queries via a DB utility or an ORM model. If additional user endpoints exist (e.g., listing users or creating new users), a **User Controller** would handle those, typically restricted to admin role.
- *Note:* In the current application, there is no public user registration endpoint (user accounts are intended to be created by an administrator or via seeding). If needed, an admin can add users by directly inserting into the DB or a future admin endpoint could be added.

Product Workflow Module

- **Components:** `productController.js`, `workflowService.js` (or similarly named service), and `productModel.js` for database access.
- **Purpose:** Handles the core product upload workflow logic. This includes creating new product entries, updating product info, retrieving products, and managing state transitions as the product moves through the workflow pipeline (Draft -> Review -> Approved -> Published).
- **Responsibilities:**
- The **Product Controller** defines API endpoints for creating a product, getting product(s), updating, and initiating state changes. It delegates most complex logic to the Workflow Service.
- The **Workflow Service** contains business logic for the product lifecycle. It knows the allowed workflow states and transitions (the state machine), and enforces rules (guards) such as completeness of checklist tasks and permission checks before changing state.
- When a new product entry is created (via the API), the Workflow Service also initializes the **checklist tasks** for that product in the database. For example, it will insert a set of predefined tasks (e.g., "Add product title and description", "Upload product images", "Set pricing") into a `tasks` table, all linked to the new product and marked as incomplete. This provides a checklist that must be completed by the user.
- The product data model (in `productModel.js`) defines fields such as title, description, price, etc., as well as the current state of the product in the workflow. It also likely includes a foreign key to the user who created it (owner or creator) and possibly a field for the Shopify Product ID once published.
- **Refactored Approach:** The workflow-related logic (state transitions, validations) has been moved out of controllers into the Workflow Service. This means controllers simply parse requests and call service methods like `WorkflowService.updateState(productId, newState, currentUser)`. The service checks if the transition is allowed (per state machine rules) and if the current user's role permits it, and whether all required checklist items are done. Only then it updates the product's state in the DB. This refactoring makes it easier to adjust workflow rules in one place and to unit test the state logic.

Checklist/Tasks Module

- **Components:** `taskModel.js` and possibly `taskController.js` (or tasks handled via product controller).
- **Purpose:** Manages the checklist items (tasks) that must be completed as part of the product upload workflow.
- **Responsibilities:**
- The **Task Model** defines the structure of a checklist item (fields: id, product_id (FK), description, completed flag, maybe a category or order). It provides queries to retrieve tasks for a product or mark tasks complete.
- In this app, tasks are typically created automatically when a product is created (via Workflow Service as mentioned). The tasks represent the necessary steps to fully prepare a product for publishing.
- The **Task/Checklist Controller** (if separate, or part of product controller) provides endpoints to fetch the checklist for a given product and to update a task's status (e.g., mark as completed). This ensures the front-end can display the checklist and allow users to check items off as they finish them.
- **Refactored Approach:** Instead of embedding checklist logic directly in the product flow in an ad-hoc way, it's now treated as a first-class entity. The separation means the code clearly differentiates between product data and the checklist state. The Workflow Service uses the Task Model to verify completion status, but the tasks management (like marking an item done) can be handled by its own

functions or controller. This modular approach allows changes to the checklist (like adding new required tasks or altering descriptions) without affecting unrelated parts of the product logic.

Shopify Integration Module

- **Components:** `shopifyService.js` (or similar utility module for external API calls).
- **Purpose:** Handles communication with the Shopify Admin API to actually **create or update the product in the Shopify store** when the workflow reaches the Publish stage.
- **Responsibilities:**
- The **Shopify Service** encapsulates the details of making API calls to Shopify. This might use Shopify's REST Admin API or GraphQL API. It requires credentials (e.g., store domain, API key and password for a private app, or access token for a custom app) which are provided via environment variables for security.
- When a product's state transitions to **Published**, the Workflow Service will invoke the Shopify Service to create the product in Shopify. For example, it will send the product's title, description, price, images, etc., to Shopify via an HTTP request. On success, Shopify returns a product ID or confirmation, which the service can store in our database (e.g., updating the product record with `shopify_product_id`).
- The Shopify Service may also handle any needed transformations (for instance, formatting our product data to match Shopify API fields) and error handling (if the Shopify API call fails, the service should catch it and respond appropriately, possibly rolling back the state change).
- **Refactored Approach:** By isolating Shopify API calls in one module, the core application logic (workflow and controllers) remains separate from external API details. This makes testing easier (Shopify calls can be mocked) and if the Shopify API changes or we switch to a different method/library, we only update this module.

Database Module

- **Components:** `db/index.js` (database connection setup), and possibly migration or seed scripts (e.g., `db/schema.sql`, `db/seed.sql`).
- **Purpose:** Establishes connection to the PostgreSQL database and provides a way to execute queries or interact via an ORM.
- **Details:**
- The DB module likely uses the `pg` library (node-postgres) or an ORM (like Sequelize, Knex, or Prisma) to connect to the Postgres instance. For example, using `pg.Pool` with the connection string from environment:

```
const { Pool } = require('pg');
const pool = new Pool({ connectionString: process.env.DATABASE_URL });
module.exports = { query: (text, params) => pool.query(text, params) };
```

  This would allow other modules to import the DB and call `db.query(...)` for SQL statements.
- If an ORM is used, this module might initialize the ORM (e.g., load models and sync or apply migrations).
- **Migrations:** The project includes either a static SQL schema (in `schema.sql`) or uses migration files. The responsibility of running migrations or the schema setup can be part of this module or

done manually. The DB module ensures the database structure is up to date when the app runs (some setups might auto-run migrations on startup, while others require manual migration).

- **Seeding:** Similarly, there may be a seed routine to populate initial data (like an admin user, default roles, etc.) for development or first-time setup. This could be a separate script or integrated such that if the database is empty on first run, it inserts some defaults.

- **Refactored Notes:** The data access responsibilities are abstracted away from business logic. Controllers or services do not directly use raw connection details; they call either an ORM model method or a DB helper. This means if we need to swap out the database or modify queries, we can do so in one place. Also, having a structured schema (with migrations) ensures any schema changes are tracked and reproducible across environments.

Utilities & Common Middleware

- **Components:** Utility libraries or helper functions (e.g., for hashing passwords, formatting data, etc.), and common middleware like error handlers or request logging.
- **Purpose:** Provide shared functionality across modules.
- **Details:**
- A **password utility** might wrap bcrypt hashing and comparison, so the rest of the code doesn't directly call bcrypt but uses a helper (for example, `hashPassword()` and `verifyPassword()` functions).
- A **JWT utility** could similarly wrap creation/verification if not directly done in Auth Service.
- **Error handling middleware:** A central error handler (e.g., `errorHandler.js`) can catch `next(err)` calls from controllers/services and format a JSON error response with proper status code. This prevents duplicating try/catch in every route. For instance, if the Shopify API call fails or a database error occurs, the service might throw an error that gets caught here and translated to a 500 response (or 400 if it's a known bad input issue).
- **Logging middleware:** In development, a logging middleware (like morgan or a custom logger) could be used to log requests and responses for debugging. Not critical to workflow but helpful for maintenance.

Each module has been designed to have a single responsibility. This modular structure makes the code easier to navigate and extend. For example, if a new workflow stage is introduced or more tasks are needed, the developer can primarily focus on updating the Workflow Service (state machine config and tasks logic) and corresponding model, rather than tracing logic spread across controllers. Similarly, if we want to change authentication method, we know to look at the Auth module specifically.

API Reference Documentation

All API endpoints are documented below with their methods, routes, purpose, and example requests/responses. The API follows a RESTful style for core resources (such as products and tasks). JSON is used for request and response bodies. Status codes adhere to conventional semantics (200s for success, 400s for client errors, 500s for server errors).

Authentication & Authorization: Most endpoints (except the login) require a valid JWT access token to be provided by the client. The token must be included in the `Authorization` header using the Bearer

scheme:

```
Authorization: Bearer <token>
```

If the token is missing or invalid/expired, the server returns **401 Unauthorized**. Additionally, certain actions are restricted to users with specific roles (e.g., only an admin can publish a product). If a user's role is insufficient, the server returns **403 Forbidden**. The required roles or permissions for relevant endpoints are noted below.

Base URL: The API is served relative to the deployed server's base address. On Replit, for example, the base might be `https://<your-repl-name>.<user>.repl.co`. All routes below are relative to this base and prefixed by `/api`. (Example: `GET /api/products`).

Authentication Endpoints

- **POST** `/api/auth/login` – *User login and token retrieval.*
  Description: Authenticates a user with username and password. On success, returns a JSON Web Token (JWT) that can be used for subsequent requests.
  Request Body: JSON with user credentials. For example:

  ```
  { "username": "alice", "password": "mySecretPass" }
  ```

  Response: JSON containing the JWT and possibly user info. For example:

  ```
  {
    "token": "<jwt-token-string>",
    "user": { "id": 5, "username": "alice", "role": "editor" }
  }
  ```

  The `token` is a signed JWT. The `user` object may be returned for convenience (so the client knows the role, etc.) – this depends on implementation.
  Authentication: *No token required.* (This endpoint is public.)
  Status Codes:
- 200 OK – login successful, JWT returned.
- 400 Bad Request – missing username/password or invalid JSON.
- 401 Unauthorized – credentials are incorrect (username not found or password mismatch).

Notes: The client should store the token (e.g., in memory or localStorage if a web app) and include it in the `Authorization` header for future API calls. The token is typically a JWT containing the user's ID and role. The token may have an expiration (e.g., valid for 1 hour) – after which the user would need to log in again or use a refresh mechanism (not implemented in this app unless otherwise noted).

User & Account Endpoints

(Currently, user management is minimal. These endpoints are typically for administrative use.)

- **GET** `/api/users` – *List users.*
  Description: Retrieves a list of all user accounts. Could be used by an admin to view users.
  Authentication: Requires a valid token with admin role.
  Response: JSON array of users (id, username, role, etc.). Password hashes are **never** included. For example:

```
[
  { "id": 1, "username": "admin", "role": "admin" },
  { "id": 2, "username": "editor1", "role": "editor" }
]
```

  Status Codes: 200 on success; 401 if not authenticated; 403 if authenticated but not an admin.

- **POST** `/api/users` – *Create a new user account.*
  Description: Creates a new user. (This might be used to add accounts by an admin, if implemented.)
  Request Body: JSON with new user details, e.g.:

```
{ "username": "bob", "password": "secret", "role": "editor" }
```

  Response: The created user (sans password), e.g.:

```
{ "id": 3, "username": "bob", "role": "editor", "created_at":
"2025-09-16T...Z" }
```

  Authentication: Requires admin role (only admins can create new users).
  Status Codes: 201 Created on success; 400 if data is missing or username already taken; 401/403 if unauthorized.

Note: Depending on project requirements, the above user endpoints may or may not be active. In some cases, user accounts are only created via direct DB scripts or seeds. If these endpoints are not needed, they can be disabled or removed to reduce attack surface.

Product Workflow Endpoints

These endpoints manage the **product entries** that go through the upload workflow. Each product entry represents a product that will eventually be uploaded to Shopify once it's fully prepared and approved.

- **POST** `/api/products` – *Create a new product entry (start a workflow).*
  Description: Starts a new product upload workflow by creating a product record in the system (initially in **Draft** state with a checklist of tasks). This is typically used by content team members (e.g.,

editors) to begin preparing a new product.
Request Body: JSON with basic product info. For example:

```
{
  "title": "Cool T-Shirt",
  "description": "A cool t-shirt with unique design.",
  "price": 25.00
}
```

Fields like `title` and `description` are usually required. Optional fields (like `price`, tags, etc.) can also be included. The server may validate that required fields are present and not empty, returning 400 if validation fails.
Response: JSON of the created product entry. For example:

```
{
  "id": 42,
  "title": "Cool T-Shirt",
  "description": "A cool t-shirt with unique design.",
  "price": 25,
  "state": "Draft",
  "created_by": "alice",
  "created_at": "2025-09-16T23:00:00Z",
  "tasks": [
    { "id": 101, "description": "Write title & description", "completed":
false },
    { "id": 102, "description": "Upload product images", "completed":
false },
    { "id": 103, "description": "Set pricing details", "completed": false }
  ]
}
```

The response includes the new product's details. Note that `state` is set to `"Draft"` initially, and a list of checklist `tasks` has been generated (all in incomplete status). The `created_by` field indicates which user created it (the current user).
Authentication: Requires a valid JWT. Any authenticated user with the appropriate role (e.g., *editor* or *admin*) can create a product. (If using strict role separation, perhaps only roles tasked with product creation, like "editor", can use this endpoint. This can be enforced via role middleware if needed.)
Status Codes:

- 201 Created – on success (returns the new product data).
- 400 Bad Request – missing required fields or invalid data.
- 401 Unauthorized – if no valid token provided.

Note: On creation, the backend automatically creates the default checklist items in the `tasks` table for this product. If the Shopify integration requires any initial setup (like reserving a draft on Shopify), that could also be done here, but in our case, actual Shopify upload happens only at publish time.

- **GET** `/api/products` – *List all product workflows.*
  Description: Retrieves a list of all product entries in the workflow system, typically to display a dashboard of products and their current status.
  Response: JSON array of product summaries. Example:

```
[
  { "id": 42, "title": "Cool T-Shirt", "state": "Draft", "created_by":
"alice", "created_at": "2025-09-16T23:00:00Z" },
  { "id": 43, "title": "Fancy Mug", "state": "Review", "created_by":
"bob", "created_at": "2025-09-15T10:30:00Z" }
]
```

  By default, all products are returned. If needed, query parameters could filter results (e.g., `?state=Draft` to filter by state), though such filters are optional and would be documented if implemented.
  Authentication: Requires a valid token (any role). Typically all team members can view the list of products. If access needs to be restricted (for example, only admins see all while normal users see only items they created), that logic would be implemented in the controller/service. By default, assume all authenticated users can see all product entries for collaborative transparency.
  Status Codes: 200 on success; 401 if unauthorized.

- **GET** `/api/products/{id}` – *Get detailed info on a specific product workflow.*
  Description: Retrieves full details of a single product entry, including all its fields and the associated checklist tasks. This is used when viewing or editing a specific product's progress.
  URL Parameters: `{id}` is the product's unique ID.
  Response: JSON object of the product. For example:

```
{
  "id": 42,
  "title": "Cool T-Shirt",
  "description": "A cool t-shirt with unique design.",
  "price": 25,
  "state": "Draft",
  "created_by": { "id": 5, "username": "alice" },
  "created_at": "2025-09-16T23:00:00Z",
  "updated_at": "2025-09-17T10:00:00Z",
  "tasks": [
    { "id": 101, "description": "Write title & description", "completed":
true },
    { "id": 102, "description": "Upload product images", "completed":
false },
```

```
    { "id": 103, "description": "Set pricing details", "completed": true }
  ]
}
```

In this example, we see some tasks completed and some not. The `created_by` could include user info (depending on how the JSON serialization is done; it might just be a user ID or include username as shown for convenience).
Authentication: Requires a valid JWT (any role). If role-based viewing restrictions are in place, a user who is not associated with this product might be forbidden (403). However, typically all authenticated team members can retrieve it.
Status Codes: 200 on success; 404 Not Found if no product with that ID (or not authorized to view); 401 if not authenticated.

- **PUT or PATCH** `/api/products/{id}` – *Update product details.*
Description: Updates the editable fields of a product entry (title, description, pricing, etc.). This is used while a product is in Draft or Review states to make changes or add information. For example, after creation, an editor might fill in more details or correct something and save changes.
Request Body: JSON with fields to update. e.g.:

```
{ "description": "Updated description text.", "price": 30 }
```

Only the provided fields will be updated (partial update if using PATCH). Some fields may be restricted: for instance, changing the `state` via this endpoint is not allowed (state transitions should use the dedicated endpoint or controlled actions). The backend should ignore or reject attempts to directly set the state here.
Response: JSON of the updated product (similar to GET product response).
Authentication: Requires valid JWT. The user must have permission to edit the product. Typically, if role separation is in place, *editors* (or the original creator) can update a Draft product's details. If the product is already in Review or Approved, only certain roles might be allowed to edit (or editing might be locked except to admins or via transitioning back to Draft). The backend enforces these rules. If a user without permission attempts an edit, it returns 403.
Status Codes:

- 200 OK on success (with updated data).
- 400 Bad Request if invalid data is provided.
- 404 Not Found if the product ID doesn't exist.
- 401/403 if unauthorized.

Note: The application might enforce that once a product is Published, its data can no longer be edited through this system (since it's live on Shopify). Any edits after publishing might require a new workflow or direct Shopify edits. If an edit endpoint is called on a Published item, the service could reject it (e.g., 400 or 403) with a message that editing a published product is not allowed.

- **PATCH** `/api/products/{id}/state` – *Change the workflow state of a product (advance to next stage or revert).*
Description: Moves a product to a new workflow state, for example from Draft to Review, Review to

Approved, or Approved to Published. This is a crucial endpoint that enforces the state machine rules and triggers side effects like publishing to Shopify.
Request Body: JSON specifying the target state. For example:

```
{ "state": "Review" }
```

The server will interpret this as a request to move the product into the **Review** stage. It will check the current state of product `{id}`, and only allow the transition if it's valid (according to the allowed transitions defined in the workflow) and if the user has permission to perform it. Optionally, instead of raw state names, the API could accept an action keyword (like `"action": "submit_for_review"`), but in our implementation using the state name is straightforward.
Response: JSON of the product with its updated state (and possibly updated fields like `updated_at`). For example:

```
{
  "id": 42,
  "state": "Review",
  "updated_at": "2025-09-17T12:00:00Z",
  "tasks": [ ... ]
}
```

The response shows the product now in "Review" state. If moving to Published, the response might also include additional info like the `shopify_product_id` after a successful upload to Shopify.
Authentication & Permissions: Requires a valid JWT. **Role requirements depend on the transition:**

- Moving from **Draft -> Review** (often called "Submit for Review"): allowed for the user who created it or generally any editor role. (The assumption is that content editors can submit their work for review once done.) The server will also check that all **Draft-stage checklist tasks are completed** before allowing this.
- Moving from **Review -> Approved** ("Approve"): typically requires a higher role, e.g., a **manager or admin** role. Only an authorized reviewer can approve the product. If a user without the proper role attempts this, the server returns 403 Forbidden.
- Moving from **Review -> Draft** ("Reject/Send Back"): allowed for a reviewer/admin to send it back for rework. Possibly also allowed for an editor to voluntarily revert to Draft to make additional changes. The application can allow this transition for admins or for the original editor if needed. (The state machine configuration will indicate who can trigger it; by default, we assume admins can do it.)
- Moving from **Approved -> Published** ("Publish"): strictly requires **admin role** in our setup. This action will trigger the **Shopify upload**. Only an admin (or a designated "publisher" role) can publish to the live store. If everything is in order, the backend will create the product via Shopify API and update the state to Published. If the Shopify upload fails, the state remains in Approved and an error is returned.
- No other transitions are allowed (e.g., you cannot jump from Draft directly to Approved or Published without going through intermediate steps).

The **state transition rules and guards** are discussed in detail in the next section of this document. The backend will validate the request thoroughly: - Check that the target state is one of the allowed next states

for the current state. - Check that all required checklist items have been completed if the transition requires it (e.g., cannot leave Draft if tasks are incomplete). - Check that the current user has the authority (role) to perform this transition. If any check fails, the transition is aborted and an error is returned.

Status Codes: - 200 OK – if transition succeeded (product state changed, and any side effect like Shopify publish succeeded). - 400 Bad Request – if the requested state is invalid or the transition is not allowed from the product's current state (the error message will clarify, e.g., "Cannot transition from Draft to Published directly" or "Checklist not complete"). - 403 Forbidden – if the user's role does not permit this action. - 404 Not Found – if product ID is invalid. - 500 Internal Server Error – if something unexpected fails (for instance, a database error or Shopify API failure; the response will likely include a generic error message or a specific one like "Shopify publish failed").

Note: This endpoint is central to the workflow. In some implementations, instead of a generic `/state` endpoint, there might be distinct endpoints for each action (e.g., `/api/products/{id}/submit`, `/api/products/{id}/approve`, `/api/products/{id}/publish`). Internally, those would similarly call the workflow logic. Our documentation assumes a single endpoint for brevity, but the actual route structure can be adjusted. Regardless, the same rules apply.

- **DELETE** `/api/products/{id}` – *Delete a product entry.*
  Description: Deletes a product from the workflow system. This might be used to remove test entries or products that were aborted. **Use with caution:** deleting a product that has already been published will not remove it from Shopify (there is no automatic Shopify deletion in this action). This is primarily for cleaning up Drafts or erroneous entries.
  Authentication: Requires admin role (to prevent unauthorized removal of data).
  Response: Typically just a success message or no content. For example:

  ```
  { "message": "Product deleted" }
  ```

  Status Codes: 200 on success (or 204 No Content); 404 if not found; 401/403 if unauthorized.

Note: The deletion will also cascade or remove related checklist tasks (the database is set up to delete tasks associated with the product via foreign key constraint or the service explicitly deletes them). Again, if the product was published, an admin would separately need to remove it from Shopify if needed (this system does not automatically handle Shopify deletions).

Checklist/Task Endpoints

These endpoints allow viewing and updating the checklist tasks for a product. Often, the tasks could be fetched as part of the product detail (as shown above). Separate endpoints are useful for marking tasks complete/incomplete individually without refetching the whole product.

- **GET** `/api/products/{id}/tasks` – *Get all checklist tasks for a product.*
  Description: Returns the list of checklist items (tasks) associated with the given product (by product ID). Each task has an `id`, description, and completion status.
  Response: JSON array of tasks. Example:

```
[
  { "id": 101, "product_id": 42, "description": "Write title &
description", "completed": true },
  { "id": 102, "product_id": 42, "description": "Upload product images",
"completed": false },
  { "id": 103, "product_id": 42, "description": "Set pricing details",
"completed": true }
]
```

Authentication: Requires valid JWT (any user who can access the parent product). If product access is restricted by role or ownership, the same restrictions apply here (the endpoint should verify the user has rights to view the product's details).
Status Codes: 200 on success; 404 if product not found (or no tasks found); 401/403 if unauthorized.

- **PATCH** `/api/products/{id}/tasks/{taskId}` – *Update a checklist task's status.*
  Description: Marks a specific checklist item as complete or incomplete. This is triggered when a user checks or unchecks an item in the UI. (In practice, once a task is done it would rarely be marked undone, but the API allows toggling in case of mistakes.)
  Request Body: JSON with the updated status, e.g.:

```
{ "completed": true }
```

  Response: The updated task object, e.g.:

```
{ "id": 102, "product_id": 42, "description": "Upload product images",
"completed": true }
```

  Authentication: Requires valid JWT. The user must be allowed to modify this product's tasks. Typically, if the user can edit the product (Draft stage, and is an editor or the creator), they can also mark tasks. Reviewers or admins could potentially mark tasks as well if assisting, but generally tasks in Draft are handled by the editor. The backend may not strictly enforce who can mark tasks (assuming collaborative environment), but it could be configured to restrict (e.g., only the assignee or creator can mark done).
  Status Codes: 200 on success; 400 if invalid data; 404 if no such task (or it doesn't belong to the product ID given); 401/403 if unauthorized.

Note: Marking a task as completed might trigger some logic like if all tasks become complete, the product could be automatically flagged as ready for review (though the actual state change to Review still requires the explicit action/endpoint). The task completion endpoints do not themselves change the product state, they only update the checklist. The state transition must be done via the `/state` endpoint as above.

General Notes on API: All endpoints return JSON. Errors are typically returned with an `{ "error": "message" }` JSON body or a similar error structure, along with an appropriate HTTP status code. For

example, a 400 might return `{ "error": "Title is required" }` or `{ "error": "Invalid state transition" }`. The exact format can be standardized via the error handler.

For authentication-protected endpoints, make sure to include the `Authorization: Bearer <token>` header in your requests. Without it, you will get a 401. For role-protected endpoints, ensure the logged-in user has the required role (if not, a 403 will result).

When integrating with the front-end or API client: - After login, store the token from `/auth/login` and include it on subsequent calls. - Handle 401 responses by redirecting to login (token might be expired or invalid). - Handle 403 by showing an appropriate "not allowed" message (the user tried an action they don't have rights for). - Use the provided API responses to update UI (e.g., when a state change is successful, the product's state and possibly tasks will update).

Now that the API is described, the next sections will explain some of the internal workings (authentication mechanism, database design, and the state machine logic that enforces the workflow rules).

Authentication & Authorization (JWT and RBAC)

The application uses **JSON Web Tokens (JWT)** for stateless authentication and implements **role-based access control (RBAC)** to restrict certain actions to authorized roles. Here we document how the authentication works in detail:

JWT Authentication Flow

- **User Login:** A user (e.g., an admin or editor) logs in via the `POST /api/auth/login` endpoint, providing their credentials. The Auth Controller verifies the credentials:
- It looks up the user by username in the database.
- It uses **bcrypt** to compare the provided password with the stored password hash. If they don't match, a 401 error is returned.
- If they match, the Auth Service generates a JWT. This is done using a secret key loaded from the environment (e.g., `JWT_SECRET`). We use the standard HS256 algorithm to sign the token.
- The JWT **payload** typically includes the user's ID, username, and role. For example, the payload might look like:

```
{
  "userId": 5,
  "username": "alice",
  "role": "editor",
  "iat": 1694918400,
  "exp": 1694922000
}
```

Here `iat` is issued-at timestamp and `exp` is expiration timestamp. The expiration might be set, for instance, to 1 hour after issuance (this is configurable).

- The token is returned to the client. The server may also include the user data in the response for convenience, but the token itself is what authorizes future requests.
- **Authenticated Requests:** The client stores the token (e.g., in local storage or a cookie if configured, though here we treat it as a bearer token). For any subsequent API call, the client must send the header `Authorization: Bearer <token>`. The server's Auth Middleware will verify this:
- It checks that the header is present and properly formatted.
- It uses the JWT library (`jsonwebtoken` in Node) to verify the token's signature with the secret key. If verification fails (token tampered with or wrong secret) or if the token is expired, it rejects the request with 401.
- If verification succeeds, the decoded token payload is attached to `req.user`.
- `req.user` now contains the user's identity (e.g., id, username, role) and can be used by subsequent middleware or controllers.
- **Token Expiration and Refresh:** If the JWT includes an expiration (`exp`), once that time is reached, the token is no longer valid. The server will respond with 401 if an expired token is presented. In this application, there is **no automatic refresh token mechanism implemented** (to keep things simple). Users must log in again to get a new token after expiration. In the future, a refresh token flow could be added if needed (i.e. issuing a long-lived refresh token and short-lived access tokens).

Token Structure and Security: The JWT is a Base64-encoded string containing header, payload, and signature. It should be treated as an opaque token on the client (not parsed or trusted without verification). The secret key used to sign is stored in an environment variable (`JWT_SECRET`) and is known only to the server. This ensures that the server can always detect if a token was forged or altered. The tokens are stateless – the server does not store them in a database or session store; all info is in the token itself. This means a token cannot be easily revoked server-side (except by using short expiration or keeping a denylist), so protect the secret and tokens carefully.

Password Storage

User passwords are **never stored in plaintext**. During user creation (for example, seeding an admin user or via a registration process), the password is run through a hashing function:

```
const bcrypt = require('bcrypt');
const saltRounds = 10;
const passwordHash = bcrypt.hashSync(plainTextPassword, saltRounds);
// store passwordHash in the database, not the plain password
```

In login, we do the opposite:

```
// retrieve user by username
// ...
const match = bcrypt.compareSync(loginPassword, user.password_hash);
if (!match) {
  // password is incorrect
  // return 401 Unauthorized
}
```

This ensures that even if the database is compromised, the actual passwords are not exposed – they are protected by the one-way hash. The saltRounds (cost factor) is set such that hashing is slow enough to deter brute force (10 is a reasonable default; this can be adjusted for security).

Role-Based Access Control (RBAC)

Each user has a role attribute (for example: `admin`, `editor`, `viewer`, etc.). Roles determine what actions a user can perform: - **Admin:** Full access – can create products, edit anything, approve, publish, manage users, etc. - **Editor (or Contributor):** Can create and edit product drafts, complete checklist tasks, and submit for review. But cannot approve or publish. - **Reviewer/Manager:** (If such a role exists) Can move products from Review to Approved (approve content), and possibly send back to Draft. Might not create products. - **Viewer:** (If exists, perhaps read-only access) Can view product status but not make changes.

(The exact set of roles depends on how the organization is set up. The two primary ones mentioned are admin and a standard user/editor role. Additional roles can be configured as needed.)

Enforcement in Code: The RBAC is enforced primarily via middleware: - The `authMiddleware` runs on protected routes to ensure the request has a valid token (user is authenticated). - On top of that, for routes requiring certain roles, we use a `requireRole` middleware or inline checks. For example, in a route definition:

```
const { requireRole } = require('./middleware/roleMiddleware');
router.patch('/products/:id/state', authMiddleware, requireRole('admin'),
productController.publish);
```

This hypothetical example would ensure that only an admin can hit the publish controller. In our more general approach (PATCH state), we handle role logic inside the service because the allowed roles depend on the transition requested (the route is the same but outcome differs). In such cases, the Workflow Service itself will check `req.user.role` against the needed role for the desired new state and throw a 403 if not allowed. - The `requireRole` middleware function is typically implemented like:

```
function requireRole(role) {
  return function(req, res, next) {
    if (!req.user) {
      return res.status(401).json({ error: 'Unauthorized' });
    }
    if (req.user.role !== role) {
      return res.status(403).json({ error: 'Forbidden: Requires ' + role + '
role' });
    }
    next();
  }
}
module.exports = { requireRole };
```

This can be extended to accept an array of roles or a minimum privilege level, depending on needs (e.g., allow 'admin' or 'manager'). - Inside controllers or services, you might also see explicit checks. For example, in the Workflow Service:

```
if (targetState === 'Published' && req.user.role !== 'admin') {
  return res.status(403).json({ error: 'Only admin can publish a product.' });
}
```

This double-checks even if the route had requireRole, adding an extra layer (or handling the case where role requirement depends on data, not just route).

Permissions Summary:
- **Admin:** Can perform any action (no restrictions in code beyond needing to be authenticated and role checked). - **Editor/User:** Can create products, edit their details, complete checklist items, and submit for review. They cannot approve or publish. - **Reviewer/Manager:** (If implemented) Can approve or reject products in review, but might not publish (unless explicitly allowed). - **Unauthenticated:** No access beyond login.

These rules are clearly documented so that future developers know how to extend or modify them. For instance, if a new role "publisher" is introduced to separate publishing duty from general admin, one would update the requireRole checks for publishing actions to accept either admin or publisher.

Authentication Middleware Implementation

To illustrate, here's a simplified version of `authMiddleware.js`:

```
const jwt = require('jsonwebtoken');
const JWT_SECRET = process.env.JWT_SECRET;

function authMiddleware(req, res, next) {
  const authHeader = req.headers['authorization'];
  if (!authHeader) {
    return res.status(401).json({ error: 'No token provided' });
  }
  const token = authHeader.split(' ')[1]; // Expect "Bearer <token>"
  if (!token) {
    return res.status(401).json({ error: 'Malformed token header' });
  }
  try {
    const payload = jwt.verify(token, JWT_SECRET);
    req.user = payload;  // payload contains userId, role, etc.
    next();
  } catch (err) {
    return res.status(401).json({ error: 'Invalid or expired token' });
  }
```

```
}
module.exports = authMiddleware;
```

This middleware is applied to protected routes (either globally for all `/api` routes except `/auth/login`, or individually per router). After this runs, `req.user` is available for subsequent handlers.

One must ensure that the JWT secret is strong and kept secret. In Replit, it should be stored as a Secret (environment variable) rather than checked into code. Also, if the token's payload includes sensitive info (we only include userId and role which are fine), design accordingly. Do not include the password hash or any secret data in the token.

Session Management

Because we use JWTs, the app is **stateless** in terms of sessions – the server does not keep track of active sessions. This means: - There is no server-side logout (logout can be handled client-side by discarding the token). - To invalidate tokens, one would need to implement a token blacklist or change the `JWT_SECRET` (which invalidates all tokens, effectively logging everyone out). - For most cases, the short expiration and the requirement to log in again is sufficient.

Database Schema and Setup

The application uses **PostgreSQL** as its database. The schema is relatively straightforward, consisting of tables for users, products, tasks, and possibly a few others for tracking (like roles if separate, or a table for storing state transition logs if needed). Below is an overview of the schema, including how to set it up and apply migrations or seeds for a development environment.

Schema Design

In SQL form, the core tables are defined as follows (simplified for clarity):

```
-- Users table: stores user accounts
CREATE TABLE users (
    id SERIAL PRIMARY KEY,
    username VARCHAR(50) UNIQUE NOT NULL,
    password_hash TEXT NOT NULL,
    role VARCHAR(20) NOT NULL DEFAULT 'editor',  -- e.g., 'admin' or 'editor'
    created_at TIMESTAMP NOT NULL DEFAULT NOW()
);

-- Products table: stores product entries for the workflow
CREATE TABLE products (
    id SERIAL PRIMARY KEY,
    title TEXT NOT NULL,
    description TEXT,
    price NUMERIC(10,2),
    state VARCHAR(20) NOT NULL DEFAULT 'Draft',
```

```
    created_by INTEGER NOT NULL REFERENCES users(id) ON DELETE CASCADE,
    shopify_product_id VARCHAR(50),  -- stores the Shopify ID after publish
(optional)
    created_at TIMESTAMP NOT NULL DEFAULT NOW(),
    updated_at TIMESTAMP NOT NULL DEFAULT NOW()
    -- We could use a CHECK constraint to ensure state is one of the allowed
values ('Draft','Review','Approved','Published'), or enforce that in app logic.
);

-- Tasks table: stores checklist tasks for each product
CREATE TABLE tasks (
    id SERIAL PRIMARY KEY,
    product_id INTEGER NOT NULL REFERENCES products(id) ON DELETE CASCADE,
    description TEXT NOT NULL,
    completed BOOLEAN NOT NULL DEFAULT FALSE
    -- We could add a "UNIQUE(product_id, description)" if we want to prevent
duplicate tasks on the same product.
    -- If tasks are associated with specific workflow stages, we might add a
'stage' column, but currently not needed.
);

-- (Optional) If keeping a log of state transitions
CREATE TABLE product_state_transitions (
    product_id INTEGER REFERENCES products(id),
    from_state VARCHAR(20),
    to_state VARCHAR(20),
    changed_at TIMESTAMP DEFAULT NOW(),
    changed_by INTEGER REFERENCES users(id)
    -- Composite key or serial id could be added
);
```

A few points about the schema: - The **users** table holds user credentials and roles. `role` is stored as text (alternatively, an integer with a reference to a separate roles table could be used, but text is simple for now). An index on username (by virtue of UNIQUE) helps login lookups. - The **products** table holds the main information about the product being prepared. Key fields: - `state`: a text field indicating the current workflow state. (Acceptable values: Draft, Review, Approved, Published – the app should use consistent strings or could use a Postgres enum type). - `created_by`: a foreign key to users, indicating who created the entry. The `ON DELETE CASCADE` ensures if a user is deleted (though likely rare), their products would also be removed. In practice, we might not delete users, but it's there. - `shopify_product_id`: holds the identifier (could be Shopify product ID or GraphQL global ID) of the product once it's created on Shopify. This can be `NULL` until the item is Published. This field allows us to potentially update or reference the live Shopify product later. - Timestamps to track creation and updates. - The **tasks** table holds the checklist items. Each task is linked to a product (foreign key). If a product is deleted, all its tasks are deleted as well (cascade). Each task has a description (like "Upload images") and a completed flag. You could also include an `order` column if tasks should maintain a specific order, or a `stage` field if certain tasks belong to certain stages, but our logic can handle stage association without an explicit column by the descriptions or by context. - The **product_state_transitions** table is optional and may not be implemented initially. It's a

nice-to-have for audit logging: every time a product's state changes, you insert a row with from->to, who did it, and when. This helps in tracking the history of a product workflow (e.g., to see if something was sent back to draft multiple times, etc.). If needed, the Workflow Service would handle inserting into this log table on transitions. If not needed now, it can be added later.

Migrations and Schema Setup

For development and deployment, the database schema needs to be created and kept up to date: - If using raw SQL migrations: The project may include a file like `db/schema.sql` containing the `CREATE TABLE` statements (like above). To set up a new database, you can run this file. For example, from a psql prompt or using a tool, run: `\i db/schema.sql` (in psql) or via command line `psql -h <host> -U <user> -d <database> -f schema.sql`. This will create all the tables. - If using an ORM (such as Sequelize or Knex migrations): There will be a migrations directory (e.g., `migrations/`) with migration files. In that case, you would run the migration command. For example, with Sequelize: `npx sequelize-cli db:migrate` (after configuring the CLI). With Knex: `npx knex migrate:latest`. The documentation of the specific ORM will outline the exact steps. Check the project README or package.json scripts; often there's a script like `"migrate": "knex migrate:latest"` or similar. - Make sure the database connection parameters (host, username, password, etc.) are correctly set in environment variables or configuration so that the migration knows where to apply the schema.

Verifying the schema: After running migrations or schema.sql, you should have the three main tables (users, products, tasks) when you `\dt` (list tables in Postgres). You can test by doing a quick query, e.g., `SELECT * FROM users;` (should be empty initially aside from any seeded admin).

Seeding Initial Data

For a development or first-time setup, it's important to have at least one user to log in with (especially an admin user), and perhaps some sample workflow data to test the system. The project likely provides a seed mechanism: - There might be a `db/seed.sql` or a `seeds` folder with data insertion scripts. - If `seed.sql` exists, it could contain insertion of an admin user and maybe a sample product.

For example, a simple `seed.sql` could be:

```
INSERT INTO users (username, password_hash, role) VALUES
('admin', '$2b$10$LHbpEMDH8PnjE7/hOAFd1.Aj0r.7OAaFVqscTdQitqcBQPQ8TbmAi',
'admin');
-- password is "admin123" hashed with bcrypt cost 10
```

This would create an admin user with username "admin" and password "admin123". **(Note: Never use simple passwords in production; this is for initial development ease. Change it after first login.)**

If using an ORM, seeding might be done through a script or by a special migration. For example, Sequelize has `db/seeders` where a seeder file could insert a user. In absence of a formal seeder, one can always manually insert using a SQL client.

Checklist tasks seeding: Usually, you don't seed tasks globally, since they are tied to products. Instead, when you insert a sample product for demo, you would also insert its tasks. If a seed adds a product, it should add tasks for it:

```
INSERT INTO products (title, description, price, state, created_by)
VALUES ('Demo Product', 'This is a demo', 10.00, 'Draft', 1);
-- Assume the admin user id=1 is creating it

INSERT INTO tasks (product_id, description, completed) VALUES
((SELECT id FROM products WHERE title='Demo Product'), 'Write title &
description', FALSE),
((SELECT id FROM products WHERE title='Demo Product'), 'Upload product images',
FALSE),
((SELECT id FROM products WHERE title='Demo Product'), 'Set pricing details',
FALSE);
```

This would give a starting product with some tasks to play with. However, including a demo product in seed is optional. Often, just seeding the admin user is enough, and the admin can then log in and create actual products via the API or UI.

Running seeds: Similar to migrations, if it's a SQL file, run it on the database after schema is created. If it's an ORM seeder, run the corresponding command (e.g., `npx sequelize-cli db:seed:all`). On Replit, you might run these via the Shell tab or a temporary piece of code.

Make sure not to accidentally run seeds in a production environment in a way that duplicates data. The seed scripts should ideally check for existing data or be idempotent. For initial dev though, that's not a big issue.

Database Connection Configuration

The application expects a **connection string** or similar config for the database. This is typically provided through an environment variable named something like `DATABASE_URL`. For example:

```
DATABASE_URL=postgres://username:password@hostname:5432/database_name
```

In a development environment, this might be pointing to a localhost Postgres. On Replit, if using an external hosted DB (like a free Heroku Postgres or ElephantSQL), you would put that URL in the Replit Secrets. If Replit provides a built-in database, you might have a different mechanism (more on that in Deployment section).

The DB module (as mentioned before) uses this to connect. It's important that this secret is set; otherwise, the app won't be able to connect to the DB and will likely crash on startup.

Maintenence & Migrations

When altering the schema (e.g., adding a new field to `products` or a new table), follow these guidelines for maintainers: - **Use migrations** if at all possible, rather than manually altering schema in the database without tracking. This ensures changes are documented and can be applied consistently across dev/staging/prod. - If using a simple `schema.sql` approach (not ideal for iterative changes), update the `schema.sql` file with the new changes and keep a manual log of what changed. - Test the migration on a dev copy of the DB or a fresh database to ensure it works. - Consider data migration: e.g., if adding a non-null column to an existing table, you might need a default or to backfill existing rows.

The current schema is small and straightforward, which should be easy to maintain. The main point for future expansion might be adding more fields to `products` (like SKU, tags, etc.) or linking to other tables (categories, etc.), depending on needs as the app grows.

Deployment and Replit Setup

The application is designed to be run on Replit, which provides an online IDE and hosting environment. This section provides instructions to deploy and run the app on Replit, including environment variable configuration, database setup, and understanding the project structure on Replit.

File/Directory Structure

First, understanding the file structure helps in locating where to make changes or investigate issues. The project is organized as follows:

```
project-root/
├── .replit               # Replit configuration file (defines run command,
etc.)
├── replit.nix            # (If present) Defines the Replit nix environment for
dependencies
├── package.json          # Node.js dependencies and scripts
├── package-lock.json
├── app.js (or server.js)  # The entry point to start the Express server
├── src/                  # (if using a src folder, else controllers/, etc. may
be top-level)
│   ├── controllers/
│   │   ├── authController.js
│   │   ├── productController.js
│   │   └── ... (other controllers)
│   ├── services/
│   │   ├── authService.js
│   │   ├── workflowService.js
│   │   └── shopifyService.js
│   ├── models/
│   │   ├── userModel.js
```

```
│   │   ├── productModel.js
│   │   └── taskModel.js
│   ├── middleware/
│   │   ├── authMiddleware.js
│   │   ├── roleMiddleware.js
│   │   └── errorHandler.js
│   └── db/
│       ├── index.js        # DB connection setup
│       ├── schema.sql      # Database schema definition
│       └── seed.sql        # Seed data for initial setup (if provided)
└── (other files like README.md, possibly a .env.example, etc.)
```

Note: The exact structure might differ (for instance, some projects do not separate `src` directory, placing folders in root). But the above hierarchy shows a typical layout. On Replit, all these files are visible in the left sidebar.

Important files: - **.replit:** This is a config file used by Replit to know how to run the project. It often contains something like:

```
run = "npm install && npm start"
```

Or it might directly call `node app.js`. It might also specify a `language = "nodejs"` and other settings. Generally, you shouldn't need to modify this unless you change the start script or want to enable a debugger. - **replit.nix:** Some Replit projects use Nix to install system-level packages. For example, if we needed the `psql` command-line tool or other dependencies, they'd be listed here. If the project is set up with a Node.js template, this may not exist, or it might exist but largely handle Node. You usually don't touch this unless adding system libs. - **package.json:** Contains the list of npm packages used (like express, pg, jsonwebtoken, bcrypt, etc.), and scripts. The `start` script is used by Replit to run the app. For example:

```
"scripts": {
  "start": "node app.js",
  "dev": "nodemon app.js"
}
```

In development, you might use `npm run dev` with nodemon (auto-restart on changes), but on Replit the run button typically executes `npm start` once. - **app.js/server.js:** This is the main code that loads the Express app and starts the server. If you want to change the server port or add global middleware, that happens here. On Replit, ensure it listens on `process.env.PORT` (which Replit sets to something like 3000) and `0.0.0.0` as host. If using Express, it might be:

```
const PORT = process.env.PORT || 3000;
app.listen(PORT, () => console.log(`Server running on port ${PORT}`));
```

Replit will map this to a public URL. - **Controllers/Services/Models:** as described in Module Structure, these contain the application logic. When deploying or debugging, if something is wrong with an endpoint, you'll likely open the corresponding controller or service file.

Environment Variables on Replit

Replit provides a way to store secret environment variables via the **Secrets** tab (the lock icon in the left sidebar). **Never hardcode sensitive credentials** in the code or commit them to version control. For this app, you will need to set the following environment variables in Replit: - **DATABASE_URL:** The connection string for your Postgres database. Example format: `postgres://user:password@hostname:5432/databasename`. For local development it might be `postgres://postgres:postgres@localhost:5432/shopify_workflow` (if you have a local Postgres with those creds). For Replit, if you use a third-party DB provider, copy their provided URL. Keep this secret. - **JWT_SECRET:** A secret passphrase used to sign JWTs. This can be any random string; the longer and more random the better (e.g., use a UUID or a securely generated string). Example: `JWT_SECRET = "super-secret-jwt-signing-key"`. In production, treat this like a password. - **SHOPIFY_STORE_DOMAIN:** (if required) The myshopify domain of the Shopify store (e.g., `"mystore.myshopify.com"`). If using Shopify API. - **SHOPIFY_API_KEY:** Shopify API key for your private or custom app. - **SHOPIFY_API_SECRET:** Shopify API secret or the admin API access token. (Exact naming depends on how you implemented the ShopifyService. Some use `SHOPIFY_ACCESS_TOKEN` for the private access token which combines key/secret.) - **NODE_ENV:** (optional) You might set this to `"production"` on Replit if you want, or leave it as default. It can affect things like logging or debug behavior.

To set these: 1. Click the **Secrets** (lock icon) in Replit. 2. Add each key and value. These will be available as `process.env.KEY` in the Node app. 3. These are not visible to others if the Repl is public, but exercise caution (Replit's philosophy allows public Repls, but secrets are hidden). Do not print these values in logs.

After setting environment vars, if the app is already running, you might need to restart it to pick up new env values (stop and start or click the Run button again).

Starting the Server on Replit

When you click the **Run** button on Replit: - It will execute the command specified in `.replit` (`npm start` typically). This will: - Install any new dependencies (`npm install` is often run automatically on first run or when package.json changes). - Run the Node application (`node app.js`). - You should see in the Replit console logs indicating the server started, e.g., `Server running on port 3000`. - Replit automatically makes the web server accessible. A preview window or a URL (like `https://<repl-name>.<user>.repl.co`) will be available to access the API. You can copy that URL and, for example, do a GET request to `https://<repl-url>/api/products` (if you have the token, etc.).

Database connectivity on Replit: If your Postgres is external, ensure that the Replit environment is allowed to reach it. Some managed Postgres (like Railway, Supabase) might have IP allowlists – you might need to allow Replit's IPs or use an option that doesn't restrict. If the database is not reachable, the app might crash on startup with an error like "could not connect to server". If that happens, double-check the DATABASE_URL and any networking requirements.

Persistent storage: Replit restarts the container from time to time (if you come back after an hour or so of inactivity). The filesystem (your code) persists, but any running processes or memory do not. Since we use a dedicated Postgres (external), the data persists there. If one were to use a local database file or Replit's own storage, data might not persist reliably across restarts (unless configured). Thus, using an external Postgres DB is recommended for persistent data.

Initializing the Database on Replit

If you have not set up the database before, you must create the tables: - The easiest way is to copy the content of `schema.sql` and run it. On Replit, you could do this by opening a Postgres client. If your Postgres is accessible via web (some providers give a web console), use that. - Alternatively, you can temporarily use a node script or modify the app to run the schema. For example, you could add a route like `/init` that runs the schema queries (not recommended for production, but for a quick setup it's possible). Or use a migration tool. - If the app uses an ORM and migrations, run the migration command in Replit's shell. E.g., open the Shell tab and run `npm run migrate` (if such a script exists) or the appropriate npx command for your migration system. - Once the schema is in place, run any seeds as described. Perhaps create the admin user. This can also be done manually: for instance, in a psql client or using a tool like DBeaver or TablePlus, connect to the DB and insert an admin user (remember to bcrypt hash the password).

One advantage of Replit is you can open a shell and even use a CLI client if installed. If `psql` is not in the environment by default, you might use Replit's Packager to install it or use the Nix config to include it. But an easier method is often connecting from your local machine to the DB or using the DB provider's tools.

Running locally (for development outside Replit)

While Replit is convenient, you can also run the application on your local machine: - Install Node.js (and npm) if not already. - Clone or download the project code. - Run `npm install` to install dependencies. - Set up a Postgres database locally or use a remote one. Create a `.env` file in the project root with the same env vars needed (DATABASE_URL, JWT_SECRET, etc.). The app (if using something like dotenv) might load those, or you can just export them in your shell before running. - Run `node app.js` (or `npm start`). The server should start on the specified port (likely 3000 if not overridden). - You can then use a tool like curl or Postman to hit `http://localhost:3000/api/auth/login`, etc. - This is useful for debugging and making changes; after confirming locally, you can push changes to Replit or update the Replit instance.

Important: Keep the development and production environments in sync regarding schema. If you apply a migration locally, apply it on the production DB too.

Deployment Updates on Replit

If the code is under version control (like GitHub) and linked to Replit, you can pull changes into Replit easily. Or if you edit in Replit, ensure you commit/push changes to remote repo for backup.

When updating environment variables on Replit (such as changing the JWT_SECRET for security), remember that any existing tokens signed with the old secret will become invalid. So ideally coordinate such changes (force re-login of users).

If scaling beyond Replit (e.g., deploying to a dedicated server or another platform), be mindful of: - Replacing or replicating environment setup (ENV vars). - Using process managers or similar (on Replit, it's simple as it just runs; on a server you might use pm2 or Docker). - Updating callback URLs or endpoints if clients are using the Replit URL (they'd need to use the new domain).

For now, Replit can handle small team usage and testing. It might not be intended for heavy production use, but it's great for demonstration and development.

Workflow State Machine and Checklist Enforcement

One of the core aspects of this application is the **state machine** that governs product status transitions, and the **checklist enforcement** that ensures a product cannot move forward in the workflow until necessary tasks are completed. This section explains how the state machine is defined in the code, what the allowed transitions are, and how the code uses the checklist (tasks) and other guards to prevent invalid state changes.

Workflow States Definition

The product workflow includes the following states: - **Draft:** Initial state when a product entry is created. In Draft, the product details are being entered/edited by the content team. All required checklist tasks (e.g., adding description, images, pricing) need to be completed in this state. Only after completing them can the product be submitted for review. - **Review:** In this state, the product's content is considered complete by the editor and is awaiting review/approval by a manager or admin. The product is essentially read-only to the editor unless it gets sent back to Draft. A reviewer checks the content for quality, compliance, etc. - **Approved:** The product has been approved and is ready to be published. In some flows, you might publish immediately upon approval, but here we have a separate state to allow a final verification or scheduling of publish. Only admins can move to Published from here. - **Published:** The final state, indicating the product has been pushed to the live Shopify store. When transitioning to Published, the integration is triggered to create the product in Shopify via API. Published items are effectively "done". Further changes would not occur in this workflow (you'd start a new workflow for major edits, or handle minor edits directly in Shopify or via a similar workflow system).

These can be considered a linear progression with the possibility of a backward step from Review to Draft if needed.

Allowed State Transitions

The state machine can be represented as a set of allowed transitions (from -> to). Any attempt to go from a state to one not in its allowed list is considered invalid and will be blocked. Here is the transition table:

Current State	Allowed Next States	Conditions / Notes
Draft	Review	All Draft checklist tasks must be completed. Triggered by "Submit for Review" action by editor.

Current State	Allowed Next States	Conditions / Notes
Review	Approved, Draft	**To Approved:** Requires manager/admin role (reviewer approves). **To Draft:** (Rejection) Allowed by reviewer/admin to request changes.
Approved	Published, Review	**To Published:** Requires admin role (publishing). Triggers Shopify upload. **To Review:** Allowed if admin decides to pull back (e.g., found an issue before publishing).
Published	*(None)*	Published is a terminal state in this workflow. (No further transitions; if changes needed, a new workflow or manual action is required.)

Some clarifications: - From **Draft**, the only forward move is to **Review**. We do not allow skipping directly to Approved or Published, because the intermediate review process is required. - From **Review**, a forward move to **Approved** (approval) or a backward move to **Draft** (rejection) are allowed. Typically, if an item is rejected, it goes back to Draft so the editor can make updates. In going back to Draft, some systems might have a separate "Rejected" state, but here we simplify by just using Draft again. We could differentiate maybe by a flag or by tasks reopened, but in code it's essentially just setting state to Draft again. - From **Approved**, the forward move is to **Published**. We also allow a backward move to **Review** if necessary (for example, if at the last moment an admin decides something needs re-review or modification, they can move it back to Review). We generally wouldn't send an Approved all the way back to Draft because that would imply major changes; instead, it goes to Review and the reviewer could then decide to reject to Draft. But in our table, we could potentially allow Approved -> Draft if we wanted a direct big revert. Currently, we allow only to Review from Approved as a controlled step back. - **Published** has no allowed transitions out. Once something is published, we don't automate any further state changes. (If needed, we might add an "Archived" or "Unpublished" state in the future to handle takedowns, but that's beyond our current scope.)

The above rules are implemented in code likely as a mapping or logic in the Workflow Service. For example, the service might have an object like:

```
const allowedTransitions = {
  "Draft": ["Review"],
  "Review": ["Approved", "Draft"],
  "Approved": ["Published", "Review"],
  "Published": []  // no transitions out
};
```

When a request comes in to change state, the service will check this structure:

```
if (!allowedTransitions[currentState].includes(targetState)) {
  // Not a valid direct transition
  throw new Error(`Invalid state transition from ${currentState} to $
{targetState}`);
}
```

By throwing an error or otherwise signalling failure, the controller will catch it and return a 400 Bad Request with that message. This prevents any unsupported jumps or duplicates (e.g., someone trying to re-approve an already approved item, or publish directly from Draft via an API call hack).

Checklist Enforcement (Task Completion Guards)

The **checklist tasks** are a crucial mechanism to ensure completeness of product data. The backend enforces that certain transitions can only happen if the necessary tasks are completed: - **Draft -> Review transition guard:** Before allowing a product to move from Draft to Review, the service checks that **all associated tasks are marked completed.** If even one task is still incomplete (e.g., images not uploaded yet), the transition is rejected. This is done by querying the tasks table for that product:

```
const incompleteCount = await Task.count({ product_id: id, completed: false });
if (incompleteCount > 0) {
  return res.status(400).json({ error: "Please complete all checklist items
before submitting for review." });
}
```

Only if `incompleteCount` is 0 (meaning all tasks are done) will the state actually be updated to Review. - **Review -> Approved guard:** Depending on your process, you might have certain tasks for reviewers (for example, a checklist for review like "Check grammar", "Verify pricing", etc.). In our current design, we did not explicitly create separate tasks for the Review stage – the assumption is the review is a single action (approve or reject). Thus, the main checklist is aimed at Draft completeness. If in the future we had review-stage tasks, we could similarly enforce them before approving. For now, the primary guard for approving is the **user role** (must be admin or reviewer). - **Approved -> Published guard:** Before publishing, aside from role check (must be admin), we double-check that everything is indeed ready. By the time something is Approved, it should already have all tasks done (since that was required to get Approved) so the tasks should all be complete. We might still double-check tasks completion just in case (especially if tasks could be edited in Review, which we haven't allowed explicitly). It's a safe measure to ensure no incomplete task slips through:

```
if (targetState === 'Published') {
  const incompleteCount = await Task.count({ product_id: id, completed:
false });
  if (incompleteCount > 0) {
    return res.status(400).json({ error: "Cannot publish while tasks are
incomplete." });
  }
  // also possibly ensure state was Approved before (since only Approved ->
Published allowed)
}
```

But since the state machine already ensures the current state must be Approved to go to Published, and tasks were completed to get to Approved, this might be redundant. Still, it doesn't hurt to check. - **Review -> Draft guard:** If a reviewer/admin sends back to Draft, we might not need to enforce tasks because going

backward implies tasks need reworking anyway. Typically, if they reject it, some tasks might be marked incomplete again (e.g., "Fix description" could be a new task or re-opening an old task). Our system doesn't auto-create new tasks on rejection; it might rely on the reviewer communicating the issues offline or via comments (not in scope of this backend). So, there's no checklist guard needed for moving backward, but after going back to Draft, the editor will address whatever was wrong (maybe they'll uncheck a task or add a note, but our model doesn't support comments; this could be a future improvement).

In summary, the **primary checklist enforcement** is at the Draft->Review transition.

The tasks are created when a product is drafted, as earlier described. The actual list of tasks might be defined in an array in code, for example:

```
const defaultTasks = [
  "Write title & description",
  "Upload product images",
  "Set pricing details"
];
```

The Workflow Service on creating a product would iterate this and insert each into the tasks table linked to the product. If you need to add or change tasks (say every product also needs a "SEO check" task), you update this list. This design means all products have the same set of tasks. If in the future different product categories require different tasks, the logic can be extended (e.g., conditionally adding tasks or having a task template table).

Role-based Guards on Transitions

We touched on this in the API and Auth sections, but to summarize in context of state machine: - The **state machine config** itself does not encode roles; it encodes structural possibilities. The **Workflow Service** combines that with role checks: - If `targetState` is `Review` and `currentState` is `Draft`, allowedTransitions says yes. Now, role: likely any authenticated user who created the product or in the editors team can do this. We usually allow the person who is working on it to submit. We might enforce that only the user who created it or any user with an "editor" role can submit. If a random other non-admin user tried to submit someone else's draft, we could block it. That could be an additional check:

```
if (targetState === 'Review' && currentState === 'Draft') {
  if (req.user.role !== 'admin') {
    // ensure the product's owner is the current user
    if (product.created_by !== req.user.id) {
      return res.status(403).json({ error:
"Only the creator or admin can submit this draft" });
    }
  }
}
```

This is an example of how we might prevent users from submitting others' work, while still letting admins do anything. This detail is up to how collaborative the environment is supposed to be. - If `targetState === 'Approved'` (from Review): the service will require an admin/reviewer role. If `req.user.role` is just 'editor', it returns forbidden. - If `targetState === 'Published'`: the service ensures `req.user.role` is 'admin'. This is a strict check in our design. - If moving backwards (`Review` to `Draft` or `Approved` to `Review`): likely only admins or reviewers (the higher roles) should do this, not the original editors. This is because an editor typically wouldn't unilaterally pull their item back from review; the reviewer decides. We enforce that e.g.:

```
if (currentState === 'Review' && targetState === 'Draft') {
  if (req.user.role !== 'admin' && req.user.role !== 'manager') {
    return res.status(403).json({ error: "Only a reviewer/admin can reject back
to draft." });
  }
}
```

Similarly, Approved -> Review probably only admin (since admin is the only one who could have approved it in first place). - The code is structured such that all these checks occur before any database update or external action. This way, if any check fails, we simply respond with an error and do not change anything.

Implementation in Code

Let's outline how a state change request flows through the code: 1. **Endpoint call:** Client calls PATCH `/api/products/{id}/state` with `{ state: "NewState" }`. Suppose the product 42 is currently in Draft and NewState is "Review". 2. **Auth middleware:** Ensures the user is logged in (req.user is set). 3. **Controller:** The productController's method for state change is invoked. It might extract `newState` from `req.body.state` and the `id` from req.params. Then it calls something like `workflowService.changeState(productId, newState, req.user)`. 4. **WorkflowService.changeState:** - It fetches the product from the database (to know current state, and possibly who created it). - It checks `allowedTransitions[currentState]` to see if `newState` is allowed. - If not allowed, it throws an error or returns an error result. - If allowed, it then performs additional guards: - If `currentState === 'Draft' && newState === 'Review'`: check tasks completion (as described). - If newState requires certain role: check `req.user.role`. - If need to ensure owner on submit: check `product.created_by`. - If any of these fail, it returns an error. - If all checks pass, it proceeds to update. It might do:

```
await Product.update(productId, { state: newState, updated_at: now });
```

or if using an ORM: `product.state = newState; await product.save();`. - If `newState === 'Published'`, it will then call the ShopifyService to create the product on Shopify:

```
const shopifyId = await shopifyService.createProduct(product);
```

This may involve constructing the product data in the format Shopify expects. If that call succeeds, `shopifyId` will be the ID from Shopify. The service might then update the product again to set `shopify_product_id`:

```
await Product.update(productId, { shopify_product_id: shopifyId });
```

If the Shopify call fails (throws an error or returns an error), the service should catch that. In case of failure, it might choose to: - Rollback the state change (set state back to Approved, since publish failed). - Or leave it in Approved and just not change it at all (perhaps it set state to Published after getting confirmation). A robust implementation would only set state to Published after a successful Shopify API response. If an error occurs, it would not change the state and instead propagate an error back to the controller. - Optionally, log the transition (if using the transitions log table, insert a row with from, to, user, timestamp). - Finally, the service returns the updated product (including new state, and possibly the shopify ID if just published). 5. **Controller (continued):** The controller receives the updated product from the service. It sends a response with that product data as JSON to the client. If an error was thrown in service, the controller (or a global error handler) catches it and sends the appropriate error response.

Preventing Invalid Moves

Thanks to the above checks, the system prevents: - Skipping stages (e.g., no Draft -> Published directly, etc. If attempted, the allowedTransitions check will fail). - Revisiting stages out of order or when not allowed (e.g., cannot approve something that isn't in Review state; cannot review something already approved, etc.). - Proceeding without completion (e.g., can't leave Draft if tasks are open). - Unauthorized actions (e.g., an editor's token cannot be used to approve or publish).

Additionally, the database itself could enforce some of these rules to an extent: - We did not set a CHECK on `state` because it's a small fixed set; we rely on app logic. But one could define `state` as an ENUM in Postgres (`CREATE TYPE state_type AS ENUM ('Draft','Review',...)`) and use that for the column to ensure only valid state values are stored. However, that doesn't enforce order, just validity of value. - The app's logic is the primary guard to maintain the integrity of workflow.

Shopify Publishing Process

When transitioning to Published, a bit more detail on what happens: - The **ShopifyService** likely uses an HTTP client (could be axios, node-fetch, or Shopify's official library if added) to send a request to Shopify's Admin API. - Prior to calling, it ensures it has the necessary credentials from env: e.g., `SHOPIFY_STORE_DOMAIN`, `SHOPIFY_API_KEY`, `SHOPIFY_API_SECRET` (or token). - For example, using REST API: it might POST to `https://{API_KEY}:{API_PASSWORD}@{STORE_DOMAIN}/admin/api/2023-01/products.json` with a JSON body containing the product info (title, body_html, variants, etc.). Or using GraphQL, post to `/{store}/admin/api/2023-01/graphql.json` with a productCreate mutation. - The data sent will include fields gathered from our product entry: - Title, Description (perhaps converted to HTML if Shopify requires HTML for product body), - Price (Shopify expects variants with pricing, if we keep it simple, one variant equal to the price), - Images: if images were uploaded in our system, we need their URLs or data. This part can be complex: possibly the app might store image URLs (maybe images uploaded to Cloudinary or an S3 bucket). If we have image URLs, we can pass them to Shopify to download. If not implemented, maybe the image upload part is manual or out-of-scope. For now, assume images are

handled and we have URLs in the description or a separate field. The integration would include them. - The ShopifyService then parses the response: - If successful, Shopify returns a product object including its `id` (and other details). - We capture that `id` (Shopify product id) and return it. - If there's an error (e.g., invalid API key, or Shopify validation error), the service throws an error with that message. - The Workflow Service uses that to update our DB as mentioned. - At this point, the product is live on Shopify. The `Published` state in our system essentially means "live". If someone tries to publish again (shouldn't happen because it's already published and state machine won't allow leaving Published), it would be blocked.

It's important to note: if the Shopify call fails, the system should **not** mark it Published. One approach is to keep it in Approved and notify the admin of the failure via the error message. The admin can then try again (maybe fix whatever issue caused failure, like missing required fields or API credentials issues, and call publish again).

Example Code Snippet: State Transition Check

For clarity, here's a pseudo-code snippet combining some of the logic described:

```
// Inside workflowService.changeState(user, productId, targetState)
const product = await Product.findById(productId);
if (!product) throw new Error("Product not found");

const currentState = product.state;
if (!allowedTransitions[currentState].includes(targetState)) {
  throw new Error(`Cannot transition from ${currentState} to ${targetState}`);
}

// Role-based guard
if (currentState === 'Draft' && targetState === 'Review') {
  // Editors can submit, ensure user is owner or admin
  if (user.role !== 'admin' && product.created_by !== user.id) {
    throw new Error("Not allowed to submit this product");
  }
}
if (currentState === 'Review' && targetState === 'Approved') {
  // Only admin/reviewer
  if (user.role !== 'admin' && user.role !== 'manager') {
    throw new Error("Only an authorized reviewer can approve");
  }
}
if (currentState === 'Review' && targetState === 'Draft') {
  // Only admin/reviewer can reject
  if (user.role !== 'admin' && user.role !== 'manager') {
    throw new Error("Only a reviewer can send back to draft");
  }
}
if (currentState === 'Approved' && targetState === 'Published') {
```

```
  if (user.role !== 'admin') {
    throw new Error("Only admin can publish");
  }
}

// Checklist guard
if (currentState === 'Draft' && targetState === 'Review') {
  const pendingTasks = await Task.count({ product_id: productId, completed:
false });
  if (pendingTasks > 0) {
    throw new Error("Complete all checklist tasks before submitting for
review");
  }
}
if (currentState === 'Approved' && targetState === 'Published') {
  const pendingTasks = await Task.count({ product_id: productId, completed:
false });
  if (pendingTasks > 0) {
    throw new Error("Cannot publish: some tasks are still incomplete");
  }
}

// If all checks pass, perform the transition
await Product.update(productId, { state: targetState, updated_at: new Date() });

if (targetState === 'Published') {
  try {
    const shopifyId = await shopifyService.publishProduct(product);
    await Product.update(productId, { shopify_product_id: shopifyId });
  } catch (err) {
    // Publishing to Shopify failed, rollback state:
    await Product.update(productId, { state: 'Approved' });
    throw new Error("Shopify publish failed: " + err.message);
  }
}

// (Optional) Log the state change
await StateTransitionLog.insert({
  product_id: productId,
  from_state: currentState,
  to_state: targetState,
  changed_by: user.id
});

// Return updated product (you might refetch it with new state or construct the
object)
return await Product.findById(productId);
```

The above pseudo-code demonstrates the sequence of checks and actions. The actual code may differ (for example, using transactions to ensure consistency, especially around the publish rollback scenario).

Ensuring Data Consistency and No Invalid States

- Because of these guards, the data in the database should always reflect a valid state relative to tasks. For instance, it should never happen that `state = "Review"` while some tasks are still marked incomplete. The system doesn't automatically go and lock tasks or anything upon state change, but logically, once in Review, all tasks were done. If a product goes back to Draft, perhaps the editor might mark some tasks as not done if they need redoing, but our current model doesn't support toggling back to false unless done via the API manually. A possible practice: a reviewer could uncheck a task when sending back to Draft to signal that item needs rework. This would be a manual step via the PATCH task endpoint by the reviewer. Our policy doesn't automatically reset tasks, but we can assume collaboration handles it.
- No two users can step on each other's toes in terms of state changes at the exact same time because each request is handled sequentially by the server. However, theoretically, if two requests tried to change the state simultaneously, one might fail because the first one already changed the state. This is a race condition scenario. With our small scale, it's unlikely, but if needed, one could implement a row lock or similar. Not needed now.
- The tasks and product records are linked; thanks to foreign keys with cascade, if you delete a product, tasks go too, preventing orphan tasks.

Summary of State Machine Benefits

By having a clearly defined state machine and enforcing it in the backend: - We maintain **process integrity**: every product goes through the required steps, ensuring quality (no product can be published without review and completeness). - The **code is easier to manage** because the allowed transitions are centralized (in `allowedTransitions` structure and related logic). If we need to modify the workflow (add a new state, or allow an extra transition), we update this central logic and possibly add roles or tasks as needed, rather than chasing down scattered conditions. - It provides clear communication to the frontend/API consumers: If someone attempts an invalid move, they get a clear error message. The front-end can also use this knowledge to disable buttons for invalid transitions rather than rely solely on error responses. - **Invalid states** (like tasks incomplete but state advanced) are prevented, which means the data in our system is reliable. When something is in Review, we can trust all tasks done. When it's Published, we know it's on Shopify.

For the developer maintaining this, to adjust the workflow, they should look at the Workflow Service. For example, if tomorrow the process changes such that a product needs a second approval stage, you'd add a new state (say "QA Approved") and adjust the allowedTransitions and guards accordingly. Or if certain products could be fast-tracked (skip review if minor), you might add logic to allow that for certain conditions.

Finally, the combination of **state machine + checklist + role checks** essentially encodes the business rules of content publishing in code. Keeping them organized and documented (like here) helps ensure future changes uphold the desired rules or adapt correctly.

This concludes the back-end technical documentation for the Shopify Product Upload Workflow application. By understanding the module structure, API contracts, authentication scheme, database design, deployment process, and the internal workflow logic, a developer should be well-equipped to maintain, debug, and extend the system.
