The TSC System Access Bible: Blueprint, Installation, & Management

Version: 1.5

Target Audience: System Administrators, Developers, ICT Staff, and End Users.

Table of Contents

Chapter 1: System Blueprint & Architecture

1.1 System Overview

The **TSC System Access Management System** is a Django-based web application designed to digitize the workflow of granting staff access to internal systems (e.g., Active Directory, CRM, IFMIS). It replaces manual paper trails with an auditable, multi-stage approval pipeline.

1.2 User Roles & Hierarchy

The system relies on a rigid role hierarchy defined in the `UserRole` model.

Role	Access Level	Responsibilities
Staff	Basic	Submit access requests, view own request status.
HOD	Approver (L1)	Reviews requests from staff within their specific **Directorate**.

ICT	Approver (L2)	Technical review after HOD approval. Routes request to specific System Admins.
System Admin	Provisioner	Final executor. Grants access to specific systems (e.g., the AD Admin only sees AD requests).
Overall Admin	Oversight	View global reports, override stuck requests (Superuser status).

1.3 Approval Workflow

The lifecycle of a request flows strictly from top to bottom.

- graph TD
- Start([User Submits Request]) -->|Status: Pending HOD| HOD{HOD Approval}
-
- HOD -- Rejected --> End([Request Rejected])
- HOD -- Approved --> ICT{ICT Approval}
-
- ICT -- Rejected --> End
- ICT -- Approved --> SysAdmin{System Admin Action}
-
- SysAdmin -- Provisioned --> Access([Access Granted])
- SysAdmin -- Rejected --> End
- SysAdmin -- Revoked --> Revoke([Access Revoked])

1.4 Tech Stack

- **Backend:** Python 3.11+, Django 5.0+
- **Database:** MySQL (Production) / SQLite (Dev)
- **Frontend:** Django Templates, Bootstrap 5, Vanilla JS (AJAX)
- **Containerization:** Docker & Docker Compose
- **Reporting:** ReportLab (PDF), OpenPyXL (Excel)

Chapter 2: Installation & Deployment

2.1 Prerequisites

- Python 3.10 or higher
- MySQL Server (if running locally without Docker)
- Git

2.2 Method A: Local Development Setup

- **Clone the Repository**
  git clone <repository_url>
- cd tsc_system_access

1.

- **Environment Setup** Create a `.env` file in `tsc_system_access/tsc_system_access/` (next to `settings.py`).
  # .env content
- SECRET_KEY=your-secret-key-here
- DEBUG=True
- # Leave DB settings commented out to use default SQLite for quick testing
- # DB_NAME=tsc_access_db
- # DB_USER=root
- # DB_PASSWORD=...

2.

- **Virtual Environment & Dependencies**
  python -m venv venv
- # Activate:
- # Windows: venv\Scripts\activate
- # Linux/Mac: source venv/bin/activate
-
- pip install -r requirements.txt

3.

- **Database & Admin**
  python manage.py migrate
- python manage.py createsuperuser

4.

- **Run Server**
  python manage.py runserver
- # Access at http://127.0.0.1:8000/

5.

2.3 Method B: Docker Deployment (Recommended)

1. **Configure `.env`** Ensure your `.env` contains valid MySQL credentials matching `docker-compose.yml`.

- **Build & Run**
  docker-compose up --build -d

2.

- **Run Migrations inside Docker**
  docker-compose exec web python manage.py migrate
- docker-compose exec web python manage.py createsuperuser

3.

Chapter 3: User Manual

3.1 Staff: Submitting Requests

1. Log in using your **TSC Number**.
2. Click **"New Access Request"**.
3. Select multiple systems (e.g., Email, HRMIS) in one form.
4. Choose the **Access Level** (User/Admin/ICT).
5. Submit. You will receive an email confirmation.

3.2 HOD: Approval Dashboard

1. Log in. You will be redirected to the **HOD Dashboard**.
2. You will see requests **only** from staff in your **Directorate**.
3. Click the **(+)** icon to expand request details.
4. Click **Action** > **Approve** or **Reject**.
   - *Note: Rejection requires a comment.*

3.3 System Admins: Provisioning

1. Log in to the **System Admin Dashboard**.
2. You will **only** see requests for the system assigned to you (e.g., if you are the "Active Directory" admin, you won't see "HRMIS" requests).
3. Click **Action** > **Grant Access** once you have created the account in the actual system.
4. The row will disappear (AJAX effect) and move to the **History** tab.

Chapter 4: System Administration

This section details how to configure the system using the Django Admin interface (`/admin/`).

4.1 Managing Users & Roles (CRITICAL)

The system relies on correct role configuration to function.

1. **Create a User (`CustomUser`):**
   - Go to **Users** > **Add User**.
   - Enter TSC Number (Username) and details.
   - **Important:** Assign a `Directorate`.
2. **Assign a Role (`UserRole`):**
   - Go to **User Roles** > **Add User Role**.
   - Select the User.
   - **Select Role:**
     - **Staff:** Select their `Hod` (Manager).
     - **HOD:** Select their `Directorate` (This determines which requests they see).
     - **System Admin:** Select the `System Assigned` (e.g., Active Directory). This determines which queue they manage.
     - **ICT:** No extra fields needed.

4.2 Revoking Access

To revoke access for security reasons:

1. Go to **Admin** > **Requested Systems**.
2. Filter by user or system.
3. Select the checkboxes next to the access rights.
4. Choose Action: **"⊖ Revoke Access (Security)"**.
5. Click **Go**. This updates the status to 'Revoked' and timestamps it.

4.3 Executive Reports

1. Go to **System Dashboard** in Admin.
2. View charts of "Active Staff with Rights".
3. Click **"Export Excel"** for a full compliance report.

Chapter 5: Developer's Walkthrough (Under the Hood)

This chapter provides the technical details required to maintain and modify the system. It identifies exact file locations and code blocks for common tasks.

5.1 Project Structure

- `tsc_system_access/`: Main project configuration (`settings.py`, `urls.py`).
- `access_request/`: The core application.
  - `models.py`: Database schema definitions.
  - `views.py`: Business logic, dashboards, and email triggers.
  - `admin.py`: Customization of the Django Admin interface.
  - `signals.py`: Logic hooks (e.g., auto-creating roles, logging logins).
  - `templates/`: HTML files for the frontend.

5.2 Developer Cheat Sheet: Locations & Code Snippets

This section details exactly where to go to change specific configurations.

5.2.1 Configuring Database Credentials

Goal: Change the database connection from local SQLite to a production MySQL server.

1. **File Location:** `.env` (This file is in `tsc_system_access/tsc_system_access/`).
2. **Logic Location:** `tsc_system_access/settings.py` reads these values.

Code to Change (in `.env`):

- # Edit these lines in your .env file
- DB_NAME=tsc_access_db
- DB_USER=root
- DB_PASSWORD=your_secure_password
- DB_HOST=127.0.0.1 # Or the IP of your MySQL server
- DB_PORT=3306

Code Reference (in `settings.py`):

- # settings.py around line 85
- DATABASES = {
-     'default': {
-         'ENGINE': 'django.db.backends.mysql',
-         'NAME': os.getenv('DB_NAME', 'tsc_access_db'),
-         'USER': os.getenv('DB_USER', 'root'),

- 'PASSWORD': os.getenv('DB_PASSWORD', ''),
- # ...
- }
- }

5.2.2 Configuring Email Settings

Goal: Update the SMTP server credentials or the "From" email address.

1. **File Location:** `.env`
2. **Logic Location:** `tsc_system_access/settings.py`

Code to Change (in `.env`):

- EMAIL_HOST_USER=your_email@gmail.com
- EMAIL_HOST_PASSWORD=your_app_password
- ICT_TEAM_EMAIL=ict_team@example.com

Code Reference (in `settings.py`):

- # settings.py around line 135
- EMAIL_BACKEND = 'django.core.mail.backends.smtp.EmailBackend'
- EMAIL_HOST = 'smtp.gmail.com'
- EMAIL_PORT = 587
- EMAIL_USE_TLS = True
- EMAIL_HOST_USER = os.getenv('EMAIL_HOST_USER', '')
- # ...

5.2.3 Modifying Email Content (Subject & Body)

Goal: Change the text sent to users when a request is submitted or approved.

1. **File Location:** `access_request/views.py`
2. **Action:** Search for the `send_mail` function.

Snippet 1: Request Submission Email (`views.py` inside `request_form_view`)

- # views.py around line 105

- send_mail(
-    subject='[TSC] New System Access Request Awaiting Your Approval',
-    message=f"A new access request from {request.user.get_full_name()} ({request.user.email}) is pending review.",
-    from_email=settings.DEFAULT_FROM_EMAIL,
-    recipient_list=[access.directorate.hod_email],
-)

Snippet 2: Approval/Rejection Email (`views.py` inside `system_admin_decision`)

- # views.py around line 630
- send_mail(
-    subject=f"[TSC] Access Update for {sys_req.get_system_display()}",
-    message=f"Dear {requester.full_name},\n\nRights have been granted/updated for {sys_req.get_system_display()}.\n\nRegards,\nTSC ICT Team",
-    # ...
-)

5.2.4 Extending the Data Model (Adding Fields)

Goal: Add a new field (e.g., "Phone Number") to the Access Request form.

1. **File Location:** `access_request/models.py`
2. **Action:** Add the field to the class, then run migrations.

Step 1: Modify `models.py`

- class AccessRequest(models.Model):
-    # ... existing fields ...
-    designation = models.CharField(max_length=100)
-
-    # NEW FIELD: Add this line
-    phone_number = models.CharField(max_length=15, blank=True, null=True)
-
-    request_type = models.CharField(...)

Step 2: Update `forms.py` You must add the new field to the form definition so it appears in the UI.

```
# access_request/forms.py
class AccessRequestForm(forms.ModelForm):
    class Meta:
        model = AccessRequest
        # Add 'phone_number' to this exclude list if you DON'T want it shown,
        # OR ensure it is NOT in the exclude list to show it.
        # Alternatively, if you use 'fields = [...]', add it there.
        widgets = {
            # Add styling
            'phone_number': forms.TextInput(attrs={'class': 'form-control'}),
        }
```

Step 3: Run Migrations Run these commands in your terminal to update the database structure:

```
python manage.py makemigrations
python manage.py migrate
```

5.2.5 Adding a New System Option

Goal: Add "Zoom" or "Slack" to the list of selectable systems.

1. **File Location:** `access_request/models.py` and `access_request/forms.py`.
2. **Action:** Update the `SYSTEM_CHOICES` tuple list.

In `models.py`:

```
class RequestedSystem(models.Model):
    SYSTEM_CHOICES = [
        ('1', 'Active Directory'),
        # ... existing items ...
        ('15', 'Pydio'),
        ('16', 'Zoom'),  # <-- ADD THIS NEW LINE
    ]
```

In `forms.py`: Ensure the `SYSTEM_CHOICES` variable in `forms.py` matches the one in `models.py`.

5.3 Technical Execution Flow

This sequence diagram illustrates the lifecycle of a request from the user's browser through the server components.

- sequenceDiagram
- autonumber
- actor Staff
- participant Browser
- participant View as Django View
- participant DB as MySQL Database
- participant Email as SMTP Server
-
- Staff->>Browser: Fills Request Form & Clicks Submit
- Browser->>View: POST request to /access/submit/
-
- activate View
- View->>View: Validate Form Data
-
- View->>DB: INSERT into AccessRequest (Status: Pending HOD)
- activate DB
- DB-->>View: Confirm Save ID
- deactivate DB
-
- View->>DB: INSERT into RequestedSystem (Systems: AD, CRM...)
-
- View->>Email: Trigger "New Request" Email to HOD
- activate Email
- Email-->>Staff: Send Confirmation Email
- deactivate Email
-
- View-->>Browser: Redirect to Success Page
- deactivate View
-
- Browser-->>Staff: Display "Submitted Successfully"

5.4 Frontend & AJAX Mechanics

The dashboard uses JavaScript to handle approvals without refreshing the page. If you need to debug why a button isn't working, check here.

File Location:
`access_request/templates/access_request/system_admin_dashboard.html`

Key JavaScript Logic:

```
● // This function captures the form submission
● function handleSystemAdminDecision(event) {
●   event.preventDefault(); // Stops page reload
●
●   // ... gathers form data ...
●
●   fetch(url, {
●     method: 'POST',
●     body: formData,
●     headers: {
●       'X-Requested-With': 'XMLHttpRequest' // Tells Django this is AJAX
●     },
●   })
●   .then(response => response.json())
●   .then(data => {
●     if (data.success) {
●       // Removes the row from the table visually
●       const systemRow = document.querySelector(`tr[data-system-
  id="${data.system_id}"]`);
●       systemRow.remove();
●     }
●   });
● }
```

Chapter 6: Troubleshooting & Debugging

This chapter addresses common issues developers and admins may face during operation.

6.1 Admin Dropdowns Not Showing

Issue: When selecting "System Admin" or "HOD" in the Django Admin user creation screen, the dynamic fields (like "System Assigned" or "Directorate") do not appear.

Root Cause: The JavaScript file responsible for toggling visibility relies on specific field names that must match the HTML `name` attributes generated by Django.

Fix:

1. Ensure `static/admin/js/userrole_admin.js` is loaded. Check the browser console (F12) for the log: ✅ UserRole Admin JS Loaded.
2. Clear your browser cache.

- Verify that `UserRoleAdmin` in `admin.py` includes the media class:
  class Media:
- js = ('admin/js/userrole_admin.js',)

3.

6.2 Page Refreshes Instead of Using AJAX

Issue: When a System Admin clicks "Confirm" on a decision, the entire page reloads instead of the row fading out smoothly.

Root Cause: The JavaScript event listener is not correctly intercepting the form submission, usually because the form selector doesn't match the HTML.

Fix:

1. Open Chrome DevTools -> Console.
2. Look for the log ✅ [FORM-X] System admin form detected. If this is missing, the JS isn't finding your forms.
3. Ensure your form action URL contains `/system-admin/decision/`.
4. **Temporary Workaround:** If AJAX is critically broken, you can comment out `event.preventDefault()` in `system_admin_dashboard.html`. The action will still succeed via a standard page reload.

6.3 Email Errors (Connection Refused)

Issue: `ConnectionRefusedError` or timeout when submitting a request.

Root Cause: The application cannot connect to the SMTP server defined in `.env`.

Fix:

1. Check `.env` settings: `EMAIL_HOST`, `EMAIL_PORT`.
2. If using Gmail, ensure you are using an **App Password**, not your login password.

- **Local Debugging:** To stop sending real emails and print them to the console instead, update `settings.py`:
  # tsc_system_access/settings.py
- EMAIL_BACKEND = 'django.core.mail.backends.console.EmailBackend'

3.

Chapter 7: Security Architecture & Protocols

7.1 Authentication & Session Management

- **Custom User Model:** The system uses a `CustomUser` model keyed by `tsc_no` (TSC Number) rather than a generic username. This aligns with the organization's unique identifiers.
- **Password Hashing:** Django's default PBKDF2 password hasher is used for storage.
- **Session Security:** The `NoCacheMiddleware` ensures that sensitive dashboards are not cached by the browser, preventing unauthorized access via the "Back" button after logout.

7.2 Role-Based Access Control (RBAC)

Access is strictly governed by the `UserRole` model found in `access_request/models.py`.

- **View Protection:** All sensitive views are protected by the `@login_required` decorator.
- **Explicit Role Checks:** Inside views (e.g., `hod_dashboard`), there are explicit checks preventing horizontal privilege escalation:
  # Example Authorization Check
- user_role = UserRole.objects.filter(user=request.user, role='hod').first()
- if not user_role:
-     return redirect('user_home')
-
- **Scope Checks:**
  - **HODs** can only see requests where `request.directorate == hod.directorate`.
  - **System Admins** can only see requests where `request.system == admin.system_assigned`.

7.3 Data Protection

- **CSRF Protection:** All forms include `{% csrf_token %}` to prevent Cross-Site

Request Forgery.
- **Environment Variables:** Sensitive keys (Database passwords, Secret Key, Email Credentials) are stored in `.env` and are **never** committed to version control.

7.4 Audit Trails & Logging

- **AccessLog Model:** Every user login is recorded in the `AccessLog` table, capturing the `user`, `timestamp`, and `ip_address`.
    - *Code Location:* `access_request/signals.py` -> `log_user_login`.
- **Decision Logging:** Every approval action records the `approver_id`, `decision_date`, and `comment` directly on the `RequestedSystem` model, creating a permanent audit trail of who approved what and when.

Chapter 8: Maintenance & Performance

8.1 Database Optimization

- **Query Optimization:** The dashboards use `select_related` and `prefetch_related` to minimize database hits (solving the N+1 query problem).
    - *Example:* `requests = AccessRequest.objects.select_related('requester').prefetch_related(...)`.
- **Indexing:** The `tsc_no` field is indexed (`unique=True`) for fast lookups.

8.2 Static Files Management

- **Development:** Static files (CSS, JS) are served by Django.
- **Production:** The `whitenoise` library is configured in `middleware` to serve compressed static assets efficiently without needing a separate Nginx configuration for static files (though Nginx is recommended for high load).
    - **Command:** `python manage.py collectstatic` must be run after any CSS/JS changes in production.

8.3 Log Rotation

- **Application Logs:** The containerized application writes logs to `stdout/stderr`, which allows Docker to handle log rotation.
- **Database Logs:** MySQL logs should be rotated via standard Linux `logrotate` configuration to prevent disk space exhaustion.

Chapter 9: Disaster Recovery

9.1 Backup Strategy

Regular backups are the only defense against data loss.

- **Database Backup:** A daily `mysqldump` of the `tsc_access_db` is recommended.
- *Script Example:*
  docker exec mysql_db mysqldump -u root -p[password] tsc_access_db > /backups/tsc_backup_$(date +%F).sql
  -
- **Code Backup:** The Git repository acts as the code backup. Ensure the `main` branch is always stable.
- **Config Backup:** Keep a secure, off-site copy of your production `.env` file.

9.2 Restoration Procedure

In the event of a catastrophic failure:

- **Re-deploy Code:**
  git pull origin master
- docker-compose up --build -d

1.

- **Restore Database:**
  cat backup_file.sql | docker exec -i mysql_db mysql -u root -p[password] tsc_access_db

2.
3. **Verify Integrity:** Login as Overall Admin and check the "System Analytics" dashboard to ensure user data and request history have been restored.

○