

Yellow Teamer Body of Knowledge

Maanit P. Trivedi

Fontys University of Applied Sciences

Course Number: Advanced Cyber Security

Table of Contents

Standards and Regulations	3
Introduction	4
Standards and Regulations	7
Threat Analysis / Secure Design.....	9
Network Architecture	11
Justification of Design Choice	13
Data Flow	15
Cost Analysis	20
Implementation	23
Infrastructure	23
Resource Breakdown and Security Functions.....	24
Web Application.....	35
Index Page.....	37
Admin Index Page	41
Registration Page	46
2 Factor Authentication	54
File Upload	60
Proposed vs Final Architecture	66
Hacking Week Evaluation	70

Reflection	72
Addressing Research Questions.....	74
BoK Checklist.....	75
Must Have	75
Additional Topics.....	75

Introduction

- **Project Overview**
 - Purpose: Deploy a secure, compliant web application on Azure.
 - Scope: Architecture design, threat mitigation and monitoring.
- **Objectives**
 - Ensure data confidentiality, integrity, and availability.
 - Align with CIS and NIST security frameworks.
 - Stay within a \$150 Azure credit budget.
- **Main Research Question**
 - How can a secure web application be designed and deployed on Azure in a way that meets professional security and compliance requirements defined by CIS and NIST standards?
- **Sub Questions**
 - Which cloud-native threats pose the most significant risk to web applications, and how do professionals typically mitigate these?
 - Which architectural, configuration, and implementation decisions most significantly affect the security, maintainability, and scalability of the deployed solution?
 - How can the final implementation be evaluated to demonstrate alignment with professional security standards and compliance frameworks?
- **DOT Framework**
 - **Pattern:**

- How do I develop a state-of-the-art solution that meets professional requirements?
- Realize as an expert
- **Methods:**
 - **Library**
 - Design Pattern Research
 - To investigate well-known cloud security patterns (e.g., zero trust, defense in depth, segmentation strategies).
 - Literature Study
 - For understanding theoretical frameworks (e.g., CIA Triad, CIS Controls, NIST SP 800-53) and how they apply to cloud environments.
 - **Workshop**
 - IT Architecture Sketching
 - To design and visualize the Azure infrastructure (e.g., VNet, NSG, App Service, Key Vault, Function App, etc.).
 - Code Review
 - To evaluate the Function App or configuration scripts (e.g., Bicep, Terraform, or ARM templates) for security flaws.
 - Requirements prioritization
 - To ensure CIS/NIST controls are implemented based on threat severity and feasibility within the Azure for Students constraints.
 - Prototyping

- To build and refine the deployment, integrating feedback when validating functionality, access control, and monitoring.
- **Showroom**
 - Guideline conformity analysis
 - To check whether your final implementation complies with CIS and NIST security guidelines (e.g., mapping controls to resources).
 - Static program analysis
 - To analyze the infrastructure-as-code, scripts, or deployment templates for misconfigurations or security flaws.
 - Peer review
 - To present the architecture and documentation to others (e.g., teacher, peers) and gather feedback on the professionalism and completeness of the solution.

Standards and Regulations

1. Application Security & Secure Development Process

- **OWASP Software Assurance Maturity Model (SAMM):**

Defines practices for software security across Design, Implementation, and Verification

- **Secure Software Development Lifecycle (SDLC):**

Includes frameworks like Microsoft SDL and NIST Secure Software Development Framework (SSDF) (practices for secure coding, testing, and deployment).

- **OWASP Top 10 Proactive Controls:**

Prioritizes security controls (e.g., input validation, encryption) during development.

2. Network & Systems Security (Cloud/On-Premises)

- **CIS Critical Security Controls:**

Focus on controls such as inventory, secure configurations, and access management

- **CIS Benchmarks:**

Security configuration guidelines for hardening systems (OS, cloud platforms).

- **Cloud Security Alliance (CSA) Guidance:**

- Security Guidance for Critical Areas of Focus in Cloud Computing

- Cloud Controls Matrix (CCM) for cloud-specific controls.

- **NIST SP 800-207 (Zero Trust Architecture):**

Principles include *"assume breach," "verify, don't trust,"* and *least privilege*.

3. IoT Security

1. **CIS IoT Companion Guide:**

Adapts CIS Critical Controls for IoT ecosystems (device authentication, secure updates).

2. **CSA IoT Security Controls Framework:**

Specific controls for IoT device lifecycle management.

4. Foundational Models

1. **CIA Triad:** Confidentiality, Integrity, Availability.

2. **Parkerian Hexad:** Extends CIA with Possession, Authenticity, Utility.

Threat Analysis / Secure Design

1. Threat Actors & Motivations

Threat Actor	Motivation	Targets
Cyber Criminals	Financial gain, data theft	User credentials, payment data, SQL DB
Script Kiddies	Fun, learning	Misconfigured services (SSH/RDP)
Insiders	Malice, negligence	Sensitive data, VM access
Hacktivists	Ideological goals	Defacement, DDoS

2. Threat Scenarios & Mitigations

A. Application Layer Threats

Threat	Impact	Mitigation
SQL Injection (SQLi)	Data theft, DB compromise	- Input validation - Azure WAF (OWASP rules)
XSS/CSRF	Session hijacking, defacement	- Sanitize user inputs - Anti-CSRF tokens - HTTPS enforcement
Brute-Force Logins	Account takeover	- MFA (Azure AD) - Rate limiting - Azure Monitor alerts

B. Network Layer Threats

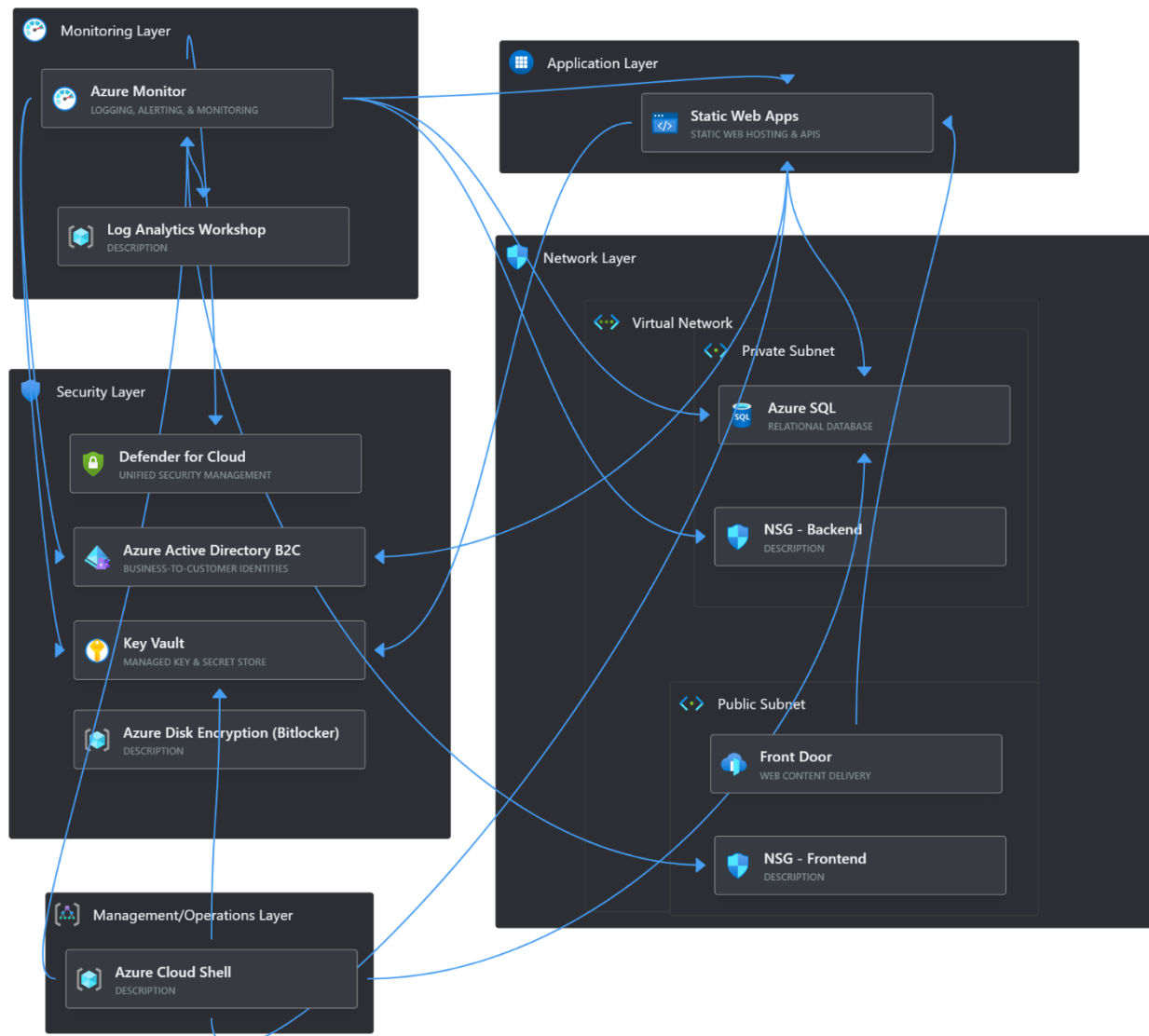
Threat	Impact	Mitigation
Exposed Ports	Unauthorized access	- NSG rules (block non-HTTP/HTTPS) - Azure Cloud Shell for VM access
DDoS Attacks	Service downtime	- Azure DDoS Protection (Basic/Standard) - Application Gateway auto-scaling
Man-in-the-Middle	Data interception	- HTTPS/TLS - HSTS enforcement

C. Data Layer Threats

Threat	Impact	Mitigation
Data Breach	Sensitive data exposure	- Azure Disk Encryption - SQL TDE (Transparent Data Encryption)
Credential Theft	Unauthorized access	- Information stored in Key Vault - Least-privilege access (RBAC)

3. Secure Design Principles

Network Architecture



- **Segmentation:**
 - **Public Subnet:** Hosts Application Gateway.
 - **Private Subnet:** Hosts Azure SQL Database (NSG restricts traffic to port 1433).
- **Encryption:**
 - **In Transit:** HTTPS via TLS 1.2+ (Azure-managed certificates).

- **At Rest:** Azure Disk Encryption (BitLocker for Windows).

B. Identity & Access Management

- **Azure AD:**
 - Enforce MFA for admin accounts.
 - Use conditional access policies.
- **Key Vault:**
 - Store database credentials, API keys, and certificates.
 - Restrict access via RBAC (Role-based access control).

C. Monitoring & Incident Response

- **Azure Monitor:**
 - Track failed logins, SQL query anomalies.
 - Set alerts for resource exhaustion (CPU/memory).
- **Microsoft Defender for Cloud:**
 - Continuously assess compliance (CIS/NIST).
 - Auto-fix vulnerabilities (e.g., unpatched OS).

Justification of Design Choice

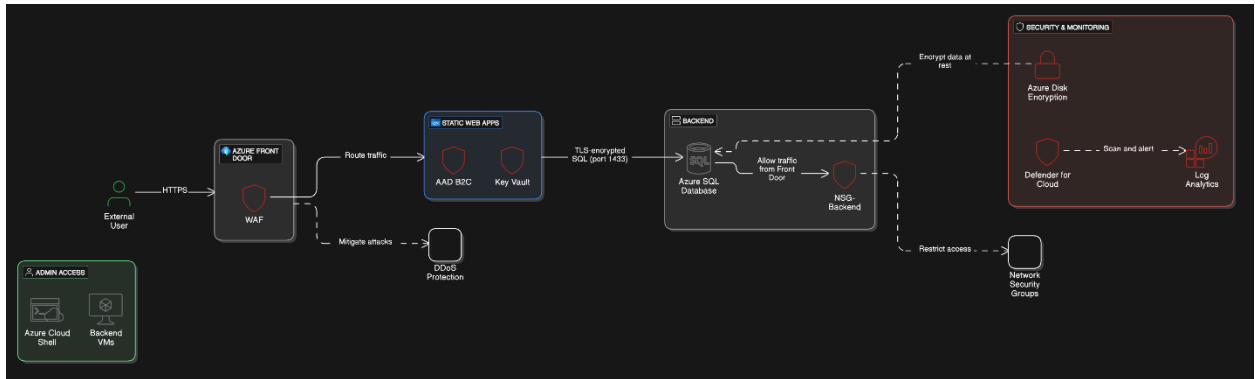
Since I have no prior experience with Microsoft, the proposed architecture was created through independent research using online documentation, official Microsoft resources, and cybersecurity best practices such as CIS Controls, NIST SP 800-53, and the Zero Trust model.

The design (shown in the image above) follows a layered approach to security. Each layer has a specific role in protecting the application's confidentiality, integrity, and availability (CIA). This layered structure helped me understand which Azure services were responsible for which security functions and how they should work together in a secure system.

- Even though I did not yet know how to implement these services in practice, I chose them based on what professionals use in real-world secure applications:
- Azure Front Door with WAF to filter incoming traffic and block common attacks like SQL injection and XSS.
- Azure Static Web Apps will host the front end with HTTPS and Entra ID (formerly Azure AD B2C) for secure login.
- Azure SQL Database is placed in a private subnet with TLS encryption to protect sensitive data.
- Azure Key Vault stores secrets like database credentials securely and prevents hardcoding.

- Azure Entra ID with MFA to handle authentication and access control.
- Network Security Groups (NSGs) to restrict traffic between the frontend, backend, and database.
- Azure Monitor and Defender for Cloud to collect logs, monitor attacks, and report compliance issues.
- Azure Disk Encryption to protect stored data in the database.
- Azure Cloud Shell to allow secure admin access without exposing remote desktop or SSH ports.

Data Flow



Initial Access Path

1. **External User Entry Point:** The data flow begins when an external user initiates a connection to the application via the HTTPS protocol. This ensures that all communication is encrypted, meets the BoK requirement for encryption in transit, and protects user credentials and data from interception.
2. **Azure Front Door with WAF:** All incoming traffic first passes through Azure Front Door, which acts as the entry point and load balancer. The Web Application Firewall (WAF) filters malicious requests by applying OWASP rule sets, blocking threats such as SQL injection and cross-site scripting (XSS). This step supports the BoK topics of input validation, system hardening, and protection against web application attacks.
3. **DDoS Protection:** Azure DDoS Protection monitors traffic patterns to detect and mitigate Distributed Denial of Service (DDoS) attacks. This service adds resilience to the system and supports the principle of failing safely, as it ensures the app remains functional during abnormal traffic spikes.

4.

Application Processing

4. **Static Web Apps:** After passing security checks, traffic routes to the Static Web Apps service, which hosts the web application front-end. There are two key security components:
 - **AAD B2C (Azure Active Directory B2C):** Handles user authentication and identity management, implementing the multi-factor authentication and conditional access policies mentioned in the BOK's Identity & Access Management section.
 - **Key Vault:** Sensitive information such as API keys and database connection strings is securely stored in Key Vault, protecting against credential theft and ensuring compliance with secure key management practices.
5. **TLS-encrypted SQL Connection:** The application then communicates with the backend Azure SQL Database using TLS-encrypted connections specifically through port 1433, implementing the "encryption in transit" security principle.

Backend Processing

6. **Azure SQL Database:** This component stores application data with multiple security layers:
 - The diagram shows controlled access from the Front Door (implementing the least privilege principle)

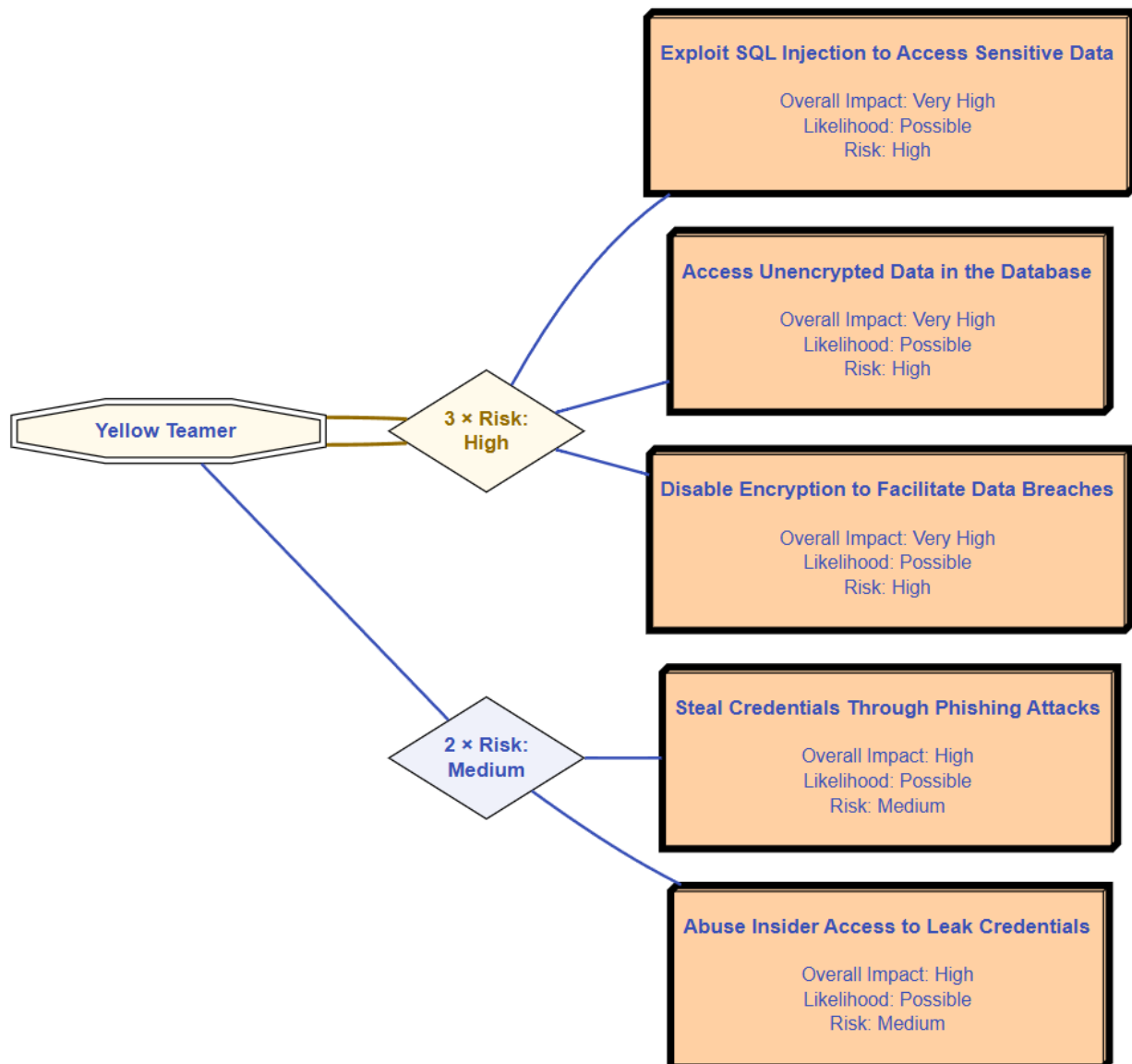
- The connection uses TLS encryption (protecting data in transit)
 - Access to the database is controlled by Network Security Groups (NSGs), enforcing the Principle of Least Privilege by only allowing traffic from known services (e.g., the web app).
7. **Network Security Groups:** These function as virtual firewalls, restricting access between different components of the architecture, and implementing the network segmentation principle.

Security & Monitoring Layer

8. **Security Monitoring Flow:** As data flows through the system, multiple security monitoring components are active:
- **Azure Disk Encryption:** Protects data at rest in the database.
 - **Defender for Cloud:** Continuously scans the environment for vulnerabilities and misconfigurations, ensuring compliance with CIS and NIST standards.
 - **Log Analytics:** Collects and analyzes logs for anomalies and security events.
9. **Administrative Access:** Administrative tasks are only performed through Azure Cloud Shell or secure backend virtual machines. This ensures that remote access is controlled, encrypted, and

not publicly exposed, following best practices for system security, least privilege, and fail-safe access control.

4. Attack Tree



5. Misuse Case Examples

Misuse Case 1: SQLi Attack

- **Actor:** Cyber **Criminal**
- **Goal:** Exfiltrate user data via SQLi.
- **Mitigation:**
 - WAF with OWASP rules.
 - Log and alert on abnormal SQL queries.

Misuse Case 2: Insider Threat

- **Actor:** Disgruntled Employee
- **Goal:** Leak database credentials.
- **Mitigation:**
 - Key Vault access audits.
 - RBAC with least privilege.

6. Compliance Alignment

Standard	Controls Implemented
CIS Controls	<ul style="list-style-type: none">- Control 3 (Vulnerability Management)- Control 4 (Admin Privileges)- Control 13 (Data Protection)
NIST SP 800-53	<ul style="list-style-type: none">- AC-2 (Account Management)- SC-28 (Encryption)- SI-4 (Monitoring)

Cost Analysis

1. Monitoring Layer

Service	Cost/Month	Notes
Azure Monitor	\$0	Basic metrics and alerts are free.
Log Analytics Workspace	\$5–10	Cost depends on data ingestion (assume 1–2 GB/month).
Defender for Cloud	\$0	Free tier covers basic vulnerability scanning and compliance checks.

2. Security Layer

Service	Cost/Month	Notes
Azure AD B2C	\$0–20	Free for ≤50,000 monthly active users (MAUs). Scale if user base grows.
Azure Key Vault	~\$0.10	Changes apply per 1,000 transactions (assume low usage).
Azure Disk Encryption	\$0	Free (uses Azure Key Vault).

3. Application Layer

Service	Cost/Month	Notes
Static Web Apps	\$0–10	Free tier for small apps; costs scale with traffic and serverless APIs.

4. Network Layer

Service	Cost/Month	Notes
---------	------------	-------

Virtual Network	\$0	Free (no cost for VNet itself).
Azure SQL Database	\$5–15	Basic tier (DTU model) for small databases.
Azure Front Door	\$25–50	WAF v2 tier included (required for OWASP rules).
NSG (Frontend/Backend)	\$0	Network Security Groups are free.

Total Estimated Monthly Costs

Layer	Cost Range
Monitoring	\$5–10
Security	\$0.10–20
Application	\$0–10
Network	\$45–105
Management/Operations	\$0.02-10
Total	\$50.12–145.10

Cost Adjusting Measures

1. Prioritizing Free Tiers:

- Azure AD B2C (≤50,000 users).
- Static Web Apps (free for small projects).

2. Optimize WAF Rules:

- Use rule exclusions to reduce false positives and unnecessary blocks.
- Enable rate limiting to reduce traffic spikes.

3. Auto-Shutdown VMs:

- Schedule VM shutdowns during off-hours (if using VMs outside Static Web Apps).

4. Monitor Log Analytics:

- Keep data ingestion ≤ 5 GB/month to minimize costs.

Implementation

Infrastructure

This diagram represents the final Azure resource setup used to host and secure the web application. The architecture was optimized for security, monitoring, and performance, while staying within the scope of Azure's student plan and the project's BoK requirements.

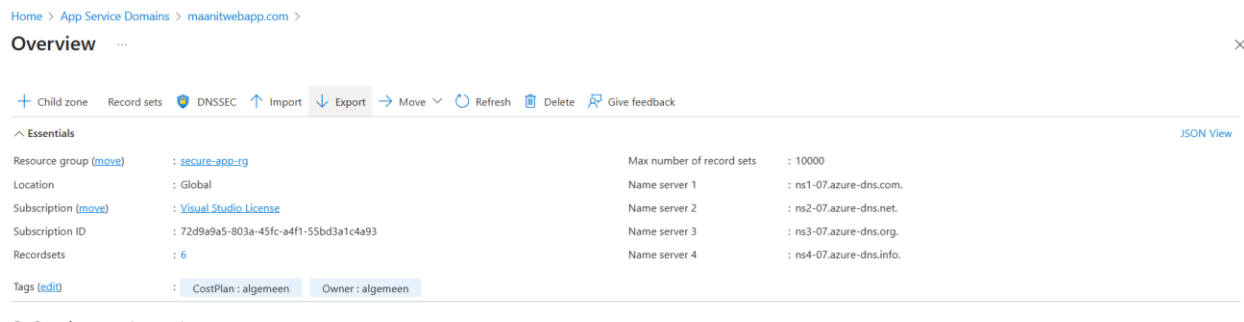
The system includes secure web delivery, backend services, role-based access, file upload protection, and centralized logging and monitoring.



Resource Breakdown and Security Functions

Web Access and Domain Management

- maanitwebapp.com (App Service Domain and DNS Zone)
 - This resource hosts the registered domain name and manages DNS records. It ensures secure HTTPS routing and domain configuration.



The domain maanitwebapp.com was purchased and managed through Azure App Service Domains. It is connected to Azure DNS, where secure name server delegation is handled through the following Azure DNS endpoints:

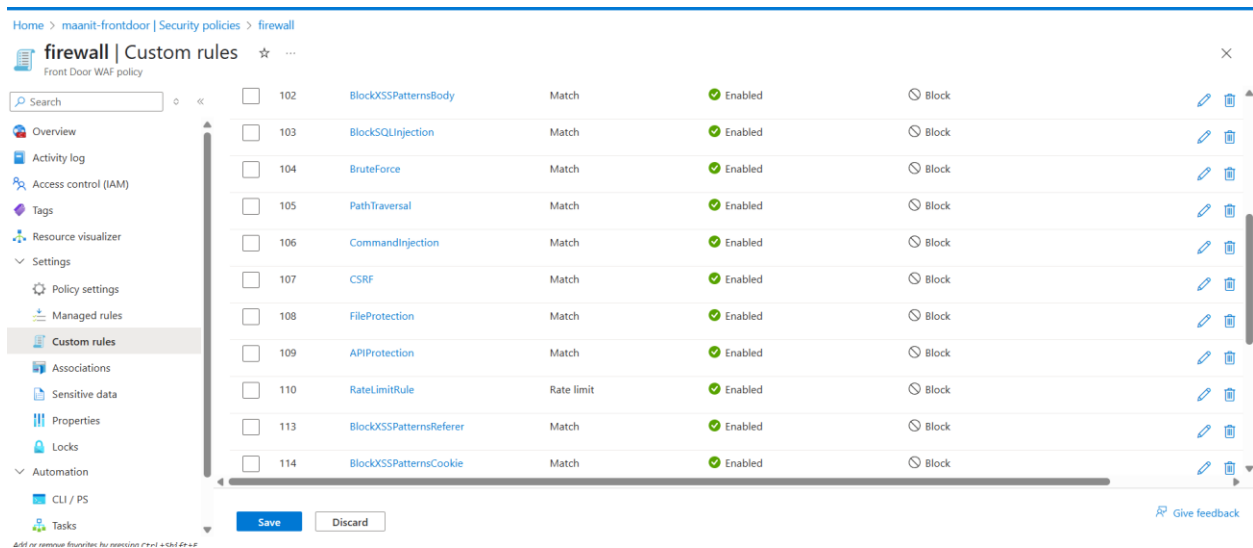
- ns1-07.azure-dns.com
- ns2-07.azure-dns.net
- ns3-07.azure-dns.org
- ns4-07.azure-dns.info

This DNS configuration ensures:

- Full control over subdomain routing
- Secure HTTPS enforcement through Azure-managed certificates
- DNSSEC-capable infrastructure (visible in the portal)

Traffic Entry and Protection

- maanit-frontdoor (Azure Front Door)
 - This service acts as the main entry point for all web traffic. It enforces HTTPS and routes incoming requests to the App Service. It also uses a WAF policy (named "firewall") to block common web threats such as SQL injection and cross-site scripting.



These are some of the WAF Rules enabled on Azure Front Door. The main goal of these rules are to block malicious requests and log the suspicious behavior (which can be seen in the dashboard)

```
maani@Maanit MINGW64 ~ (main)
$ curl -i "https://maanitwebapp.com/?input=;ls"
% Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
           %    0     0              Dload  Upload   Total   Spent    Left   Speed
100  17  100   17    0    0    31      0 --:--:-- --:--:-- --:--:--   31HT
TP/2 403
date: Wed, 21 May 2025 22:20:09 GMT
content-type: text/html
content-length: 17
x-azure-ref: 20250521T222009Z-r1749659b862qvx2hc1PARTz3c0000000f40000000005d42
x-cache: CONFIG_NOCACHE
Nice try, buddy.
```

A simulated attack was performed to validate the effectiveness of the CommandInjection WAF rule configured on Azure Front Door. Based on the following results:

Alert counts by site and risk

This table shows, for each site for which one or more alerts were raised, the number of alerts raised at each risk level.

Alerts with a confidence level of "False Positive" have been excluded from these counts.

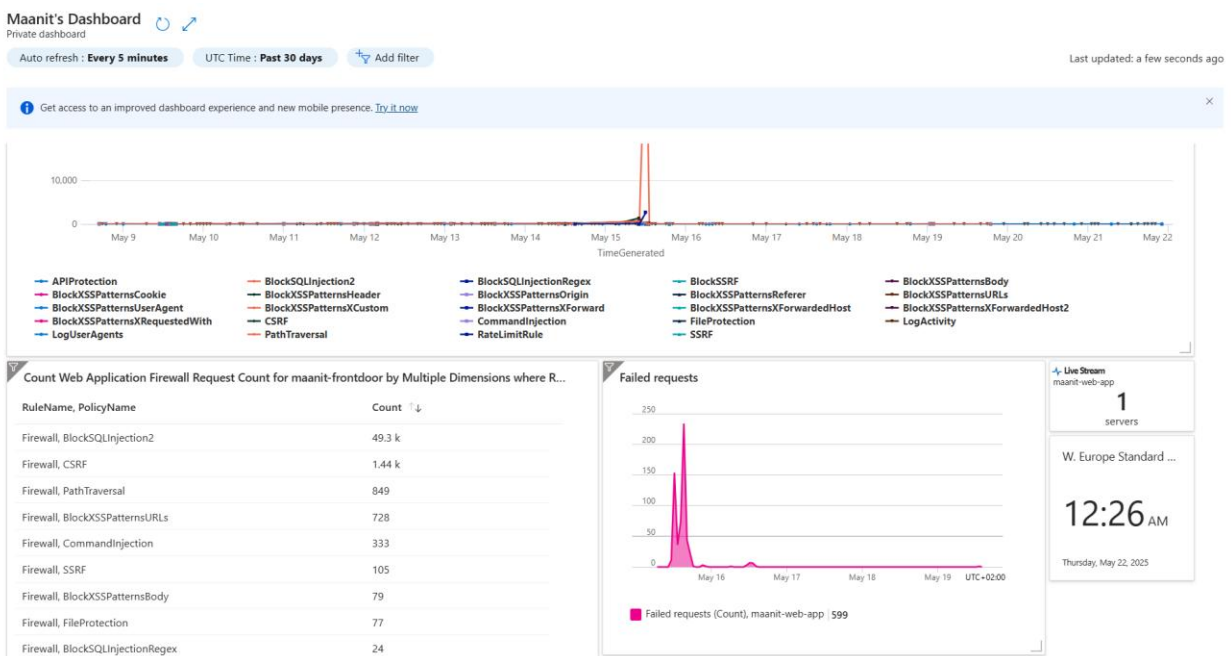
(The numbers in brackets are the number of alerts raised for the site at or above that risk level.)

		Risk			
		High	Medium	Low	Informational
		(= High)	(>= Medium)	(>= Low)	(>= Informational)
Site					
	https://maanitwebapp.com	0	4	0	5
		(0)	(4)	(4)	(9)

To validate the security of the live web application after deployment, I conducted a dynamic analysis using OWASP ZAP.

Monitoring and Logging

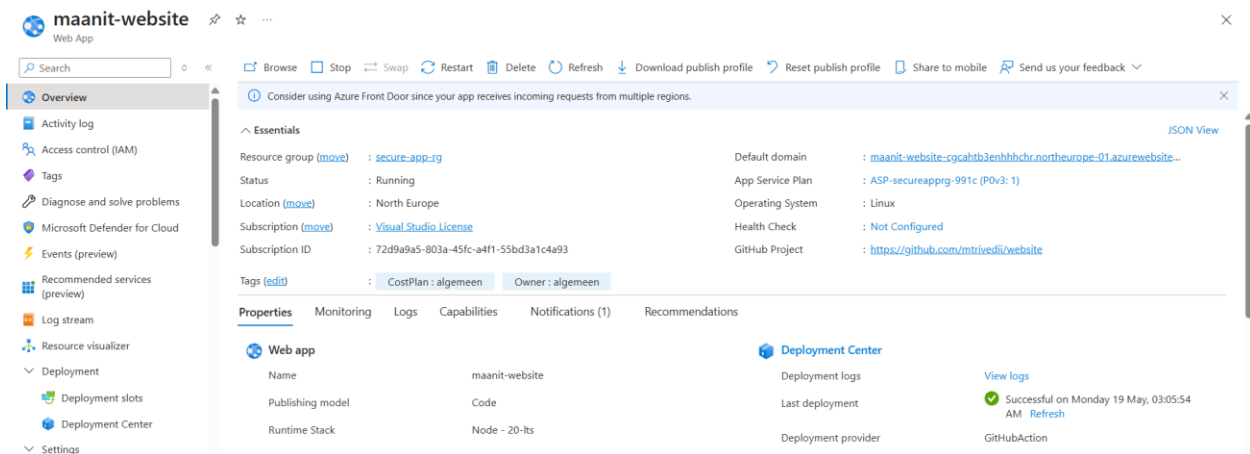
- maanit-logs (Log Analytics Workspace)
 - This is the central location for collecting logs from multiple services.
- Application Insights (maanit-web-app)
 - Provides performance monitoring and error tracking for the application.
- Behavior and Security Insights (Solutions)
 - Collect usage and security-related events.
- Shared Dashboards
 - Used to visualize log data and detect abnormal activity.



Here is the dashboard that was created to monitor the malicious activity on the web app. This Azure dashboard combines monitoring for security and performance in real time. It includes data from both Application Insights and Front Door WAF logs, with queries and charts built from Log Analytics.

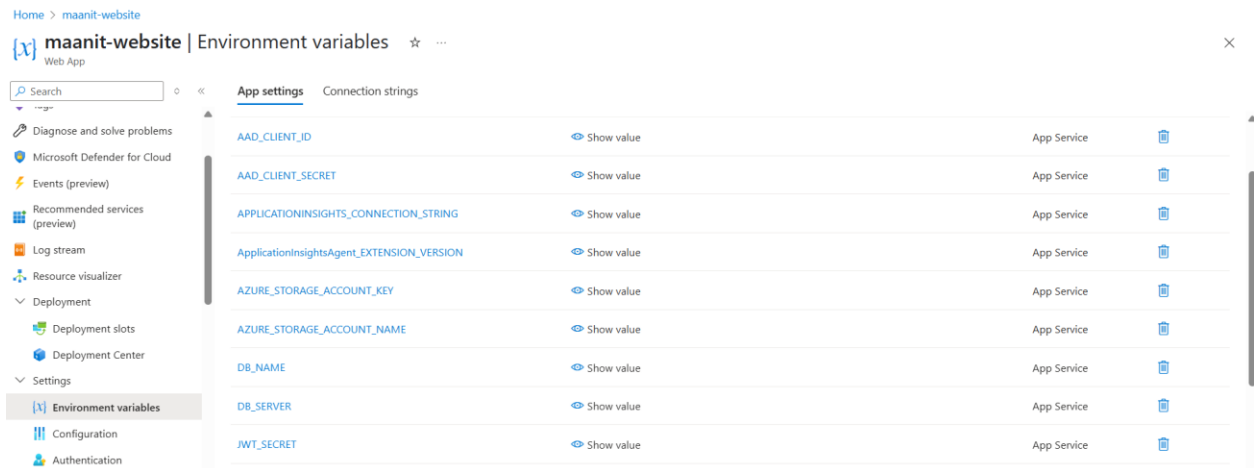
Web Application Hosting

- maanit-website (Azure App Service)
 - Hosts both the frontend and backend of the application. Security features include 2FA using TOTP, secure cookie settings, input validation, and HTTPS enforcement.
- secureappgrga106 (Azure Blob Storage)
 - Used for file upload functionality. Security measures include SAS token authentication, file extension validation, and upload sanitization.



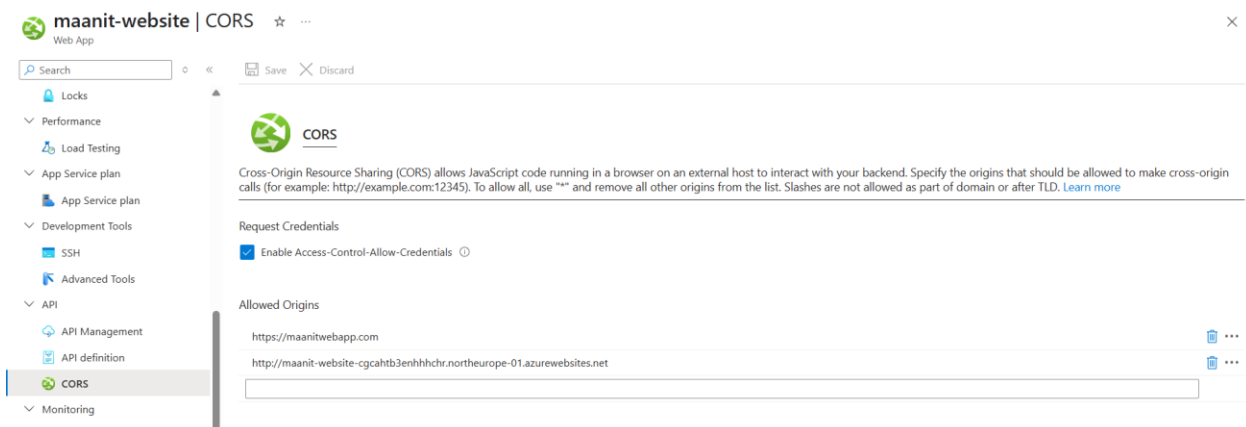
The maanit-website App Service hosts both the frontend and backend of the web application using a Node.js 20 LTS runtime on a Linux environment. It is deployed using GitHub Actions as part of a secure CI/CD pipeline.

- Deployed through GitHub Actions, ensuring traceability and secure automation
- HTTPS is enabled by default



Here are some of the app settings that are used to securely inject secrets into the backend without hardcoding them:

- Azure Blob Storage access keys
- SQL database name and server
- JWT signing secret
- Application Insights configuration
- Entra ID (AAD) credentials

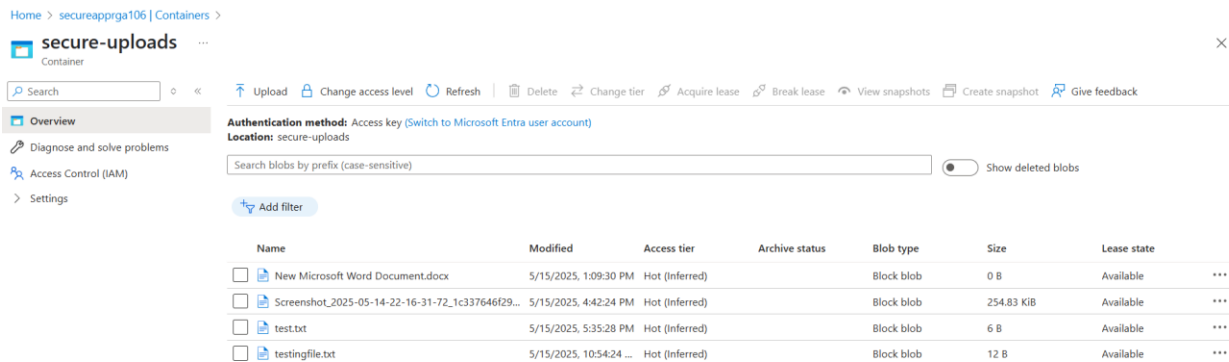


CORS is explicitly restricted to:

- <https://maanitwebapp.com>

- The Azure default domain

Credentials (cookies, headers) are allowed only from these origins, which prevents unauthorized cross-site API calls.

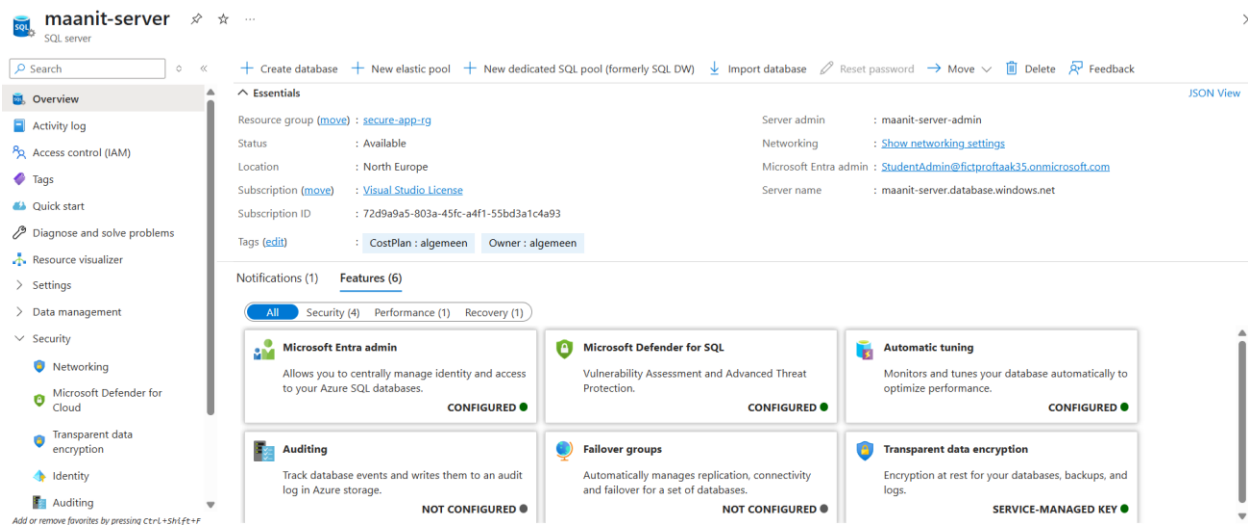


This storage account container is used for file uploads handled via SAS tokens from the backend. Files are uploaded directly from the client without exposing storage credentials.

- The container is set to private, ensuring files are not publicly accessible.
- The authentication method is configured via access key, and SAS tokens are scoped to specific uploads.
- Supported MIME types and extensions are validated in `getSasToken.js`.
- Files uploaded here are:
 - Sanitized for filename and content type
 - Blocked if they are scripts or executable content
 - Limited in size
 - Uploaded using time-limited SAS URLs generated server-side

Database Layer

- maanit-server (Azure SQL Server)
 - Hosts the application's SQL databases.
- maanit-server/maanit-db
 - The primary application database. It is accessed using Azure Managed Identity and protected with Transparent Data Encryption (TDE). Access is further restricted by firewall rules.
- maanit-server/master
 - A default system database, not used directly by the application.



As seen in the SQL Server overview, Transparent Data Encryption is enabled with a service-managed key. This ensures all data at rest, including backups and transaction logs, is encrypted automatically.

- Microsoft Entra ID is integrated and configured for secure identity-based access.
- Role-based access is used within the app (admin/user), while Azure manages secure platform-level authentication.
- Database roles are enforced using the Role column in the users table and used in JWT tokens.

- Azure services are allowed as an exception for integrated services (like App Service).
- Defender for SQL is enabled, providing vulnerability scanning and threat detection.

Home > Recent > maanit-db (maanit-server/maanit-db) > maanit-server

maanit-server | Networking

SQL server

Search

Overview
Activity log
Access control (IAM)
Tags
Quick start
Diagnose and solve problems
Resource visualizer
Settings
Data management
Security

Microsoft Defender for Cloud
Transparent data encryption
Identity

Allow certain public internet IP addresses to access your resource. [Learn more](#)

+ Add your client IPv4 address (31.201.252.114) + Add a firewall rule

Rule name	Start IPv4 address	End IPv4 address
ClientIPAddress_2025-05-19_13-38-08	145.93.37.197	145.93.37.197
ClientIPAddress_2025-5-12-1-34-18	31.201.252.114	31.201.252.114
maanit-func	20.107.224.53	20.107.224.53
query-editor-aa6a2a	145.93.37.152	145.93.37.152
query-editor-dec826	31.201.252.114	31.201.252.114
query-editor-fa8468	145.93.37.152	145.93.37.152

Exceptions

☒ Allow Azure services and resources to access this server

Only specific IP addresses (developers and services) are allowed to access the database.

Welcome | SQLQuery_2 - (65) m...ft.com

Run | Cancel | Disconnect | Change | Database: maanit-db | Estimated Plan | Enable Actual Plan | Parse | Enable SQLCMD | To Notebook

```

1 SELECT TOP (1000) [id]
2     ,[email]
3     ,[password]
4     ,[AzureID]
5     ,[Role]
6     ,[twoFactorSecret]
7     ,[twoFactorEnabled]
8     ,[last_login]
9     ,[failed_login_attempts]
10    ,[account_locked]
11    ,[lockout_until]
12    ,[mfa_last_verified]
13    ,[twoFactorRecoveryCodes]
14    ,[registration_complete]
15    ,[status]
16    ,[twoFactorTempSecret]
17 FROM [dbo].[users]

```

Results | Messages

	id	email	password	AzureID	Role	twoFactorSecret	twoFactorEnabled
8	21	maanit49@gmail.com	\$2b\$10\$0awjju4VLug5rvouMj1Zuo6RnGB7oTGqP.c.drmX00mVGQNL...	NULL	admin	FFWDUNDZEU3VU5SMNBHMKCDFYSFIQRW	1
9	22	test6@test.com	\$2b\$10\$ojxmzbkbfFc1EL6L48LK0QAzvcr6vYtF/RHA0waK56T3v1Swb...	NULL	user	PBIUWULKJGGVTLXHXJTCJT3FBSPWNCI	1
1...	23	mahmoud@gmail.com	\$2b\$10\$iWZa5Bv1agf500bGFA.pCu1Z1v2K9FAI668/GG0cf77MrA8WB3...	NULL	user	NULL	0
1...	24	test7@test.com	\$2b\$10\$i0rvsfvIc.yhvmz4cg35Yuj8T4n38a.MRY8fHuYHv14hNAyxo...	NULL	user	HSOVOSTOKRRICIQT2OFMEEYLWOSISISKP	1
1...	25	testlukasmaanitweb@gmail.com	\$2b\$10\$AVuMK06N1250/bvxhHcjjuuc6TTjeAHq1WaePDVW4pg4At..Ewa...	NULL	user	NULL	0
1...	26	test12345@test.com	\$2b\$10\$5tLL0yQ7P/.uVsxJH0QMDhu9tItVFQxcCTDIiG5PthFMZHRMdc...	NULL	user	MRTG0IK6ERWCMKSFPRPEMST5NFXHMQRS	1
1...	27	a@sudo-s.nl	\$2b\$10\$djBj2t4a403jg0Q7WtbVG.e45QqhJWtCFrEqrtZe61PokJAZV/...	NULL	user	N52VIXKYGVES6TJZJF4S65CSNB3VWVCL	1
1...	28	govas48604@inkight.com	\$2b\$10\$d.rQgvd751ff2bh5DD19J.ok4DU2hXQXhKkGQv.uCgJfPmVBFf...	NULL	user	NULL	0
1...	29	test@testingthetester.com	\$2b\$10\$4RdwZSuCVsCAPVba73.v0.tt83wLbt154k6naLrBr1x74yF1Da...	NULL	user	NULL	0
1...	30	nifiv96173@jaziipo.com	\$2b\$10\$et33.WPHMgCrZUuvNgYec.ezeBbt3BSEqPyr.eddtJrCW1Ry.BZ...	NULL	user	MIQVKS300M2FGJBGQYD4PBOLBWVUKY	1
1...	31	test@test.nl	\$2b\$10\$hm7x2XbcdNHZ6npjJKMzguspaJqPhR7YwAcDBW2EV1Irh3JFwI...	NULL	user	NEUMEL3RJ34VGRZKHFTHO45SKKZWTORLP	1
1...	32	seggybenito10@gmail.com	\$2b\$10\$8DInuX314pNm/1NoXeh6/.bxqVL41N1ismZjAlS3pzhf1YyS...	NULL	user	NULL	0
1...	33	risole8653@neuraxo.com	\$2b\$10\$A71KEhp024LQ738wvqcPezk9Z16DuCAHp.yJ0sQuBvEqy487...	NULL	user	GR4VMIJIME75U4BXEFHGE23TOAYVM23I	1

The users table which showcases the accounts made on the web app confirms:

- Passwords are hashed (e.g., \$2b\$10... → bcrypt)

- 2FA secrets and recovery codes are stored securely
- Role field is clearly separated (admin, user)
- Login attempt tracking and lockout fields (failed_login_attempts, lockout_until) are implemented

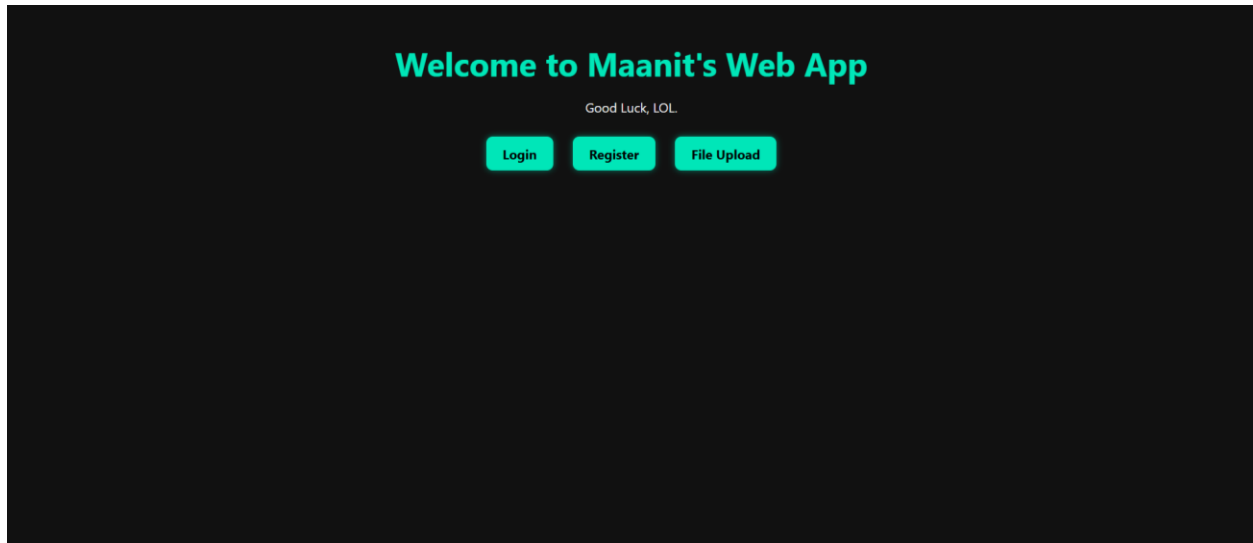
Justification of Infrastructure Adjustments

Component	Initial Architecture	Final Architecture	Justification
Frontend Hosting	Azure Static Web Apps	Azure App Service (Linux)	Static Web Apps lacked backend flexibility and secure secret injection. App Service allowed full backend control and TOTP integration.
Authentication	Azure AD B2C	Custom TOTP + bcrypt + JWT	B2C was too restrictive. Custom solution allowed MFA with more control over session logic.
Secrets Management	Azure Key Vault	App Service Environment Variables	Vault removed for simplicity in development. Env vars acceptable for non-production; vault recommended for scaling/rotation.
Traffic Entry Point	Azure Front Door + WAF	Same (Azure Front Door + WAF)	Implemented with over 20 WAF rules. Supported protection against OWASP Top 10 threats like SQLi, XSS, and SSRF.
DDoS Protection	Platform-level (assumed via Front Door)	Same (implicitly via Azure infrastructure)	Azure handles DDoS at the platform level. No custom configuration needed, aligning with the “Fail safely” principle.
Encryption (In Transit)	HTTPS (TLS)	Fully implemented	HTTPS is enforced throughout the system.

Encryption (At Rest)	Azure Disk Encryption (BitLocker)	SQL Transparent Data Encryption (TDE)	TDE is native and sufficient for Azure SQL. BitLocker was irrelevant for PaaS.
Backend Hosting	Azure SQL in Private Subnet with NSG	Azure SQL with firewall + TDE + Defender	Private subnet removed due to App Service deployment model. IP-based restrictions are used instead.
Network Segmentation	NSGs (Frontend/Backend)	Not applicable	App Service doesn't support NSGs; perimeter protection is handled by Front Door + WAF.
Monitoring & Logging	Azure Monitor, Log Analytics, Defender	Fully implemented	Dashboard built with live telemetry, WAF logs, and security alerts.
Admin Access	Azure Cloud Shell or Backend VM	Azure Cloud Shell only	Removed backend VM to reduce complexity and attack surface. Aligned with secure access controls.
DevSecOps	GitHub Actions + CodeQL	Same (fully implemented)	Secure CI/CD pipeline enforced code quality and compliance.
Dynamic Testing	Not specified	OWASP ZAP DAST added	DAST ensured that vulnerabilities were caught post-deployment.
Incident Response	Microsoft Sentinel (Planned)	Not implemented fully	Sentinel covered in In Depth Research

Web Application

Index Page



This is the landing page of my web application (<https://maanitwebapp.com>)

Security Features

- HTTPS Implementation:
 - The site is served over HTTPS, ensuring all traffic is encrypted in transit
- Session Management:
 - No pre-authentication cookies visible, preventing common tracking vulnerabilities
- Limited Entry Points:
 - Only three controlled entry points (Login, Register, File Upload). Users logged in as admin have access to additional pages which are showcased in the landing page after logging in

Authentication Flow

- Separation of Authentication Methods:
 - Distinct buttons for Login and Register enforce proper user flows
- MFA Integration:
 - The login button connects to the 2FA implementation described in the code
- Registration Process:
 - Features strong password requirements and the requirement to set up additional authentication.

File Upload Security

- Controlled File Upload:
 - A separate button for file upload functionality implements the security controls from `getSasToken.js`
- Content Validation:
 - Uploads are validated for both extension and content-type as specified in the code
- Azure Blob Integration:
 - Files are stored in Azure Blob storage with properly scoped SAS tokens

Code Explanation

HTML Structure & Security Features

```
1. <!-- public/index.html -->
2. <div class="container">
3.   <h1>Welcome to Maanit's Web App</h1>
4.   <p>Good Luck, LOL</p>
5.
6.   <div class="button-container">
7.     <a href="/login.html" class="button" id="login-link">Login</a>
8.     <a href="/registration.html" class="button" id="register-link">Register</a>
9.     <a href="/upload.html" class="button">File Upload</a>
10.    <a href="/admin.html" class="button" id="admin-panel-link" style="display:none;">Admin
Panel</a>
11.    <button id="logout-button" class="button" style="display:none;">Logout</button>
12.  </div>
13. </div>
14.
```

- Admin-only elements are initially hidden with style="display:none;"
- Static pages provide clear entry points with limited attack surface
- Separation of pages prevents unauthorized information disclosure

Authentication State Detection

```
1. // public/index.html
2. document.addEventListener('DOMContentLoaded', function() {
3.   debugLog('Page loaded, checking auth status');
4.
5.   const logoutBtnInstance = document.getElementById('logout-button');
6.   if (logoutBtnInstance) {
7.     logoutBtnInstance.addEventListener('click', logout);
8.   }
9.
10.  // Check token on page load to set up initial UI
11.  checkToken();
12. });
13.
14. function checkToken() {
15.   const token = localStorage.getItem('auth_token');
16.
17.   // Default to non-authenticated UI state
18.   if(logoutButton) logoutButton.style.display = 'none';
19.   if(adminPanelLink) adminPanelLink.style.display = 'none';
20.   if(usersLink) usersLink.style.display = 'none';
21.   if(loginLink) loginLink.style.display = 'inline-block';
22.   if(registerLink) registerLink.style.display = 'inline-block';
23.
24.   // Only proceed if token exists
25.   if (token) {
26.     // Additional auth logic here...
27.   }
```

```
28. }  
29.
```

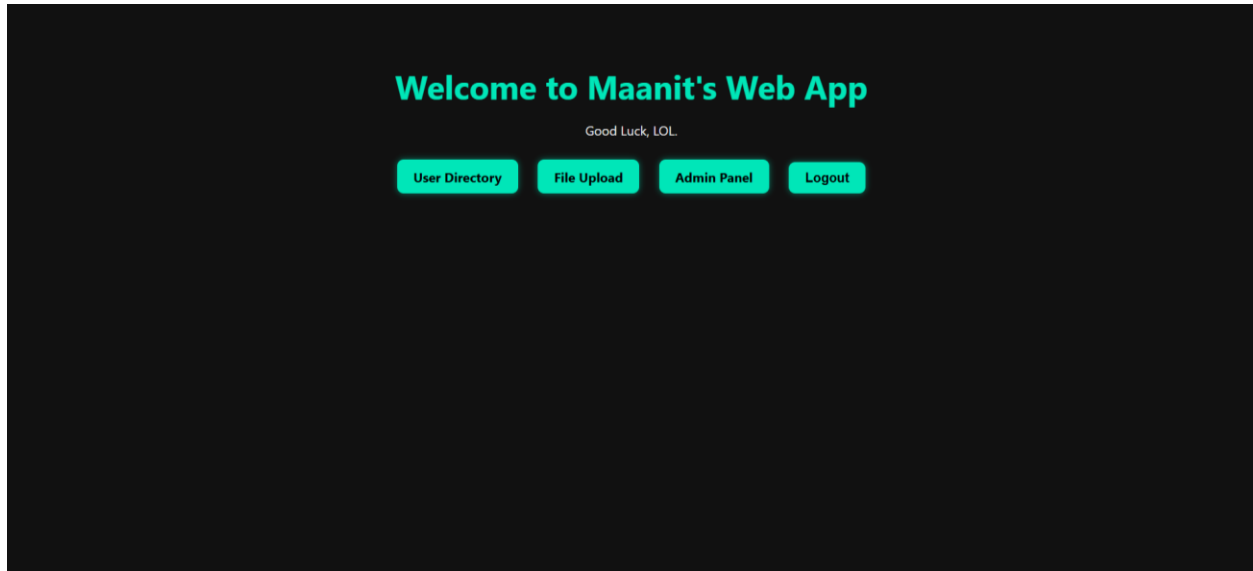
- Default state assumes no authentication
- Only reveals non-sensitive UI elements to unauthenticated users
- Avoids client-side session detection spoofing through proper defaults

Route Protection for Unauthenticated Users

```
1. // public/staticwebapp.config.json  
2. {  
3.   "routes": [  
4.     {  
5.       "route": "/upload.html",  
6.       "allowedRoles": ["authenticated"],  
7.       "statusCode": 401,  
8.       "serve": "/401.html"  
9.     },  
10.    {  
11.      "route": "/",  
12.      "allowedRoles": ["anonymous", "authenticated"],  
13.      "serve": "/index.html",  
14.      "statusCode": 200  
15.    }  
16.  ]  
17. }  
18.
```

- Server-side route protection prevents URL manipulation attacks
- Unauthenticated users attempting to access protected resources are redirected to 401.html
- The home page explicitly allows both anonymous and authenticated users

Admin Index Page



This is the homepage when the admin logs in. The authenticated home page demonstrates role-based access control in action, displaying administrative features that implement the principle of least privilege.

Authentication State Management

- Session Persistence:
 - User authentication state is maintained through the JWT implementation in the code
- Login Verification:
 - The presence of admin features indicates that successful 2FA authentication has occurred.
- Privilege Identification:
 - The user's role (admin) has been properly extracted from the JWT payload. The presence of admin features indicates that successful 2FA authentication has occurred

Role-Based Access Control

- Admin Features:
 - The "User Directory" and "Admin Panel" buttons implement the role-based restrictions defined in `staticwebapp.config.json`
- Authorization Enforcement:
 - These routes are protected through the `allowedRoles` configuration, preventing unauthorized access
- Defense in Depth:
 - Server-side verification supplements client-side visibility controls

Secure Session Management

- Logout Functionality:
 - Explicit logout button implements proper session termination
- Token Invalidation:
 - Clicking logout executes the code that clears both `localStorage` and cookies
- Client-Side Cleanup:
 - Removes the JWT token from browser storage as seen in the `index.js` implementation

Code Explanation

Authentication State & Role-Based Access Control

```
1. function checkToken() {  
2.   const token = localStorage.getItem('auth_token');  
3.   // ... Element references ...  
4.  
5.   if (token) {  
6.     // ... Show authenticated UI elements ...  
7.  
8.     const parts = token.split('.');  
9.     if (parts.length === 3) {  
10.      const payload = JSON.parse(atob(parts[1]));  
11.      if (payload.role && payload.role.toLowerCase() === 'admin') {  
12.        // Show admin UI elements  
13.      }  
14.    }  
15.  }  
16. }  
17.
```

(index.js)

This is the core client-side security implementation that:

- Checks if the JWT token exists in browser storage
- Parses the token to extract the payload data (JWT tokens are base64-encoded with three parts: header, payload, signature)
- Examine the role property to determine if the user is an admin
- Conditionally renders different UI elements based on the user's role
- Implements the "Principle of Least Privilege" by only showing admin features to users with admin roles

Route Protection in Configuration

```
1. {
2.   "routes": [
3.     {
4.       "route": "/users.html",
5.       "allowedRoles": ["authenticated", "admin"],
6.       "statusCode": 401,
7.       "serve": "/401.html"
8.     },
9.     // ... other routes ...
10.  ]
11. }
12.
```

(staticwebapp.config.json)

This is server-side protection that:

- Defines access control rules at the routing level
- Creates a security boundary for sensitive pages
- Redirects unauthorized users to a 401 page
- Provides defense-in-depth by protecting routes even if client-side checks are bypassed

Logout Functionality

```
1. function logout() {
2.   localStorage.removeItem('auth_token');
3.   localStorage.removeItem('user_role');
4.   localStorage.removeItem('user_email');
5.   document.cookie = 'auth_token=; expires=Thu, 01 Jan 1970 00:00:00 UTC; path=/; SameSite=Lax';
6.   window.location.reload();
7. }
8.
```

(index.js)

This implements secure session termination by:

- Removing all authentication data from localStorage
- Explicitly expiring the authentication cookie
- Using a thorough approach that cleans up multiple storage locations
- Implementing the security principle of "complete session termination"
- Enforcing the "SameSite=Lax" policy for cookies as a defense against CSRF attacks

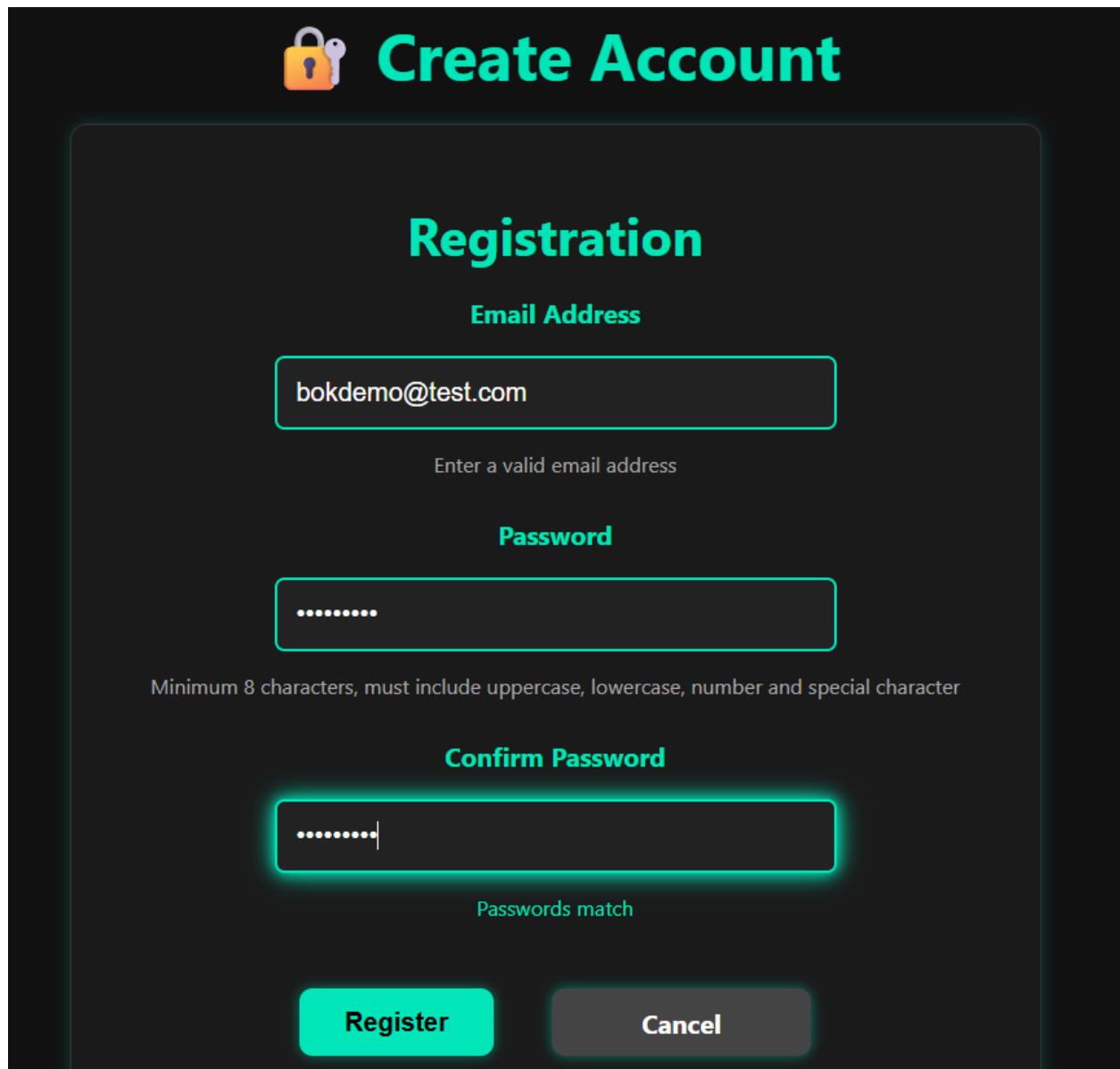
JWT Token Generation (Backend)

```
1. const finalAuthToken = jwt.sign(  
2.   {  
3.     userId: user.id,  
4.     email: user.email,  
5.     role: user.Role  
6.   },  
7.   process.env.JWT_SECRET || 'dev-secret-key',  
8.   { expiresIn: '1h' }  
9. );  
10.  
11. res.cookie('auth_token', finalAuthToken, {  
12.   httpOnly: true,  
13.   secure: true,  
14.   // ... other settings ...  
15. });  
16.
```

(login.js)

This implements secure token creation by:

- Creating a signed JWT with the user's ID, email, and role
- Including the critical role of property determines authorization decisions
- Setting a 1-hour expiration for security (limiting the attack window)
- Using httpOnly to prevent client-side JavaScript from accessing the cookie
- Using secure: true to ensure cookies only flow over HTTPS
- Creating the foundation for the entire authentication system

Registration Page

The image shows a registration page with a dark background. At the top, there is a lock icon and the text "Create Account". Below this, the word "Registration" is centered in a large, bold, light blue font. Under "Registration", the label "Email Address" is centered. Below it is a text input field containing "bokdemo@test.com". Below the input field, the text "Enter a valid email address" is displayed in a smaller, light blue font. Below this, the label "Password" is centered. Below it is a text input field filled with dots. Below the input field, the text "Minimum 8 characters, must include uppercase, lowercase, number and special character" is displayed in a smaller, light blue font. Below this, the label "Confirm Password" is centered. Below it is a text input field filled with dots. Below the input field, the text "Passwords match" is displayed in a smaller, light blue font. At the bottom, there are two buttons: "Register" (light blue) and "Cancel" (dark grey).

The registration page implements multiple security best practices aligned with NIST SP 800-63B guidelines for secure authentication:

Password Security Implementation

- Strong Password Requirements:

- Enforces NIST-recommended complexity:
 - Minimum 8 characters length
 - Must include uppercase and lowercase letters
 - Requires at least one number
 - Requires at least one special character
- Password Confirmation:
 - Implements real-time verification with visual feedback ("Passwords match" indicator) to prevent user errors
- Visual Password Masking:
 - Password fields properly mask input while providing character count feedback

Input Validation & Security

- Email Validation:
 - Implements proper email format validation using `validator.isEmail()` on both client and server side
- Form Protection:
 - Hidden CSRF token generated for each form submission.
- Real-time Validation:
 - Client-side validation provides immediate feedback while maintaining server-side validation as the security boundary

Backend Security Controls

- Server-side Validation:
 - The validateRegistration middleware in register.js provides defense-in-depth by re-validating all inputs
- Rate Limiting:
 - Implements IP-based rate limiting via the rateLimit middleware to prevent brute force attacks
- Secure Password Storage:
 - Passwords are hashed using bcrypt with proper salt rounds before database storage
- Email Normalization:
 - Emails are normalized (trimmed and converted to lowercase) for consistent security checks

User Registration Flow

- Secure Redirect:
 - After successful registration, users are directed to the 2FA setup page
- Session Security:
 - Registration process includes setting up for multi-factor authentication
- Clean UI Design:
 - Minimalist interface reduces potential for social engineering by presenting only necessary information

This implementation directly addresses several CIS and NIST controls, including CIS Control 4 (Administrative Privileges) and NIST SP 800-53 AC-2 (Account Management) through its approach to secure user account creation.

Code Explanation

Client-Side Validation & CSRF Protection

```
1. <form id="registrationForm">
2.   <!-- CSRF protection token -->
3.   <input type="hidden" id="csrfToken" name="csrfToken" />
4.
5.   <div class="form-group">
6.     <label for="email">Email Address</label>
7.     <input type="email" id="email" name="email" required
8.           autocomplete="email" />
9.     <small class="hint">Enter a valid email address</small>
10.  </div>
11.
12.  <div class="form-group">
13.    <label for="password">Password</label>
14.    <input type="password" id="password" name="password" required
15.          pattern="^(?=.*[a-z])(?=.*[A-Z])(?=.*\d)(?=.*[@$!%*?&])[A-Za-z\d@$!%*?&]{8,}$"
16.          autocomplete="new-password" />
17.    <small class="hint">Minimum 8 characters, must include uppercase, lowercase, number and
18.    special character</small>
19.  </div>
```

(registration.html)

This snippet implements multiple security controls:

- CSRF protection via hidden token generation
- HTML5 form validation using the required attribute
- Complex password pattern using regex that enforces minimum length and character requirements
- Secure autocomplete attributes to help password managers without compromising security

Password Confirmation Validation

```
1. // Check password match
2. confirmPasswordInput.addEventListener('input', function() {
3.   if (passwordInput.value !== confirmPasswordInput.value) {
4.     passwordMatchHint.textContent = "Passwords do not match";
5.     passwordMatchHint.className = "hint error";
6.   } else {
7.     passwordMatchHint.textContent = "Passwords match";
8.     passwordMatchHint.className = "hint success";
9.   }
10. });
11.
```

(registration.html)

This creates real-time client-side validation that:

- Compares password fields as the user types
- Provides immediate visual feedback about password matching status
- Enhances user experience while maintaining security
- Helps prevent accidental submission of mismatched passwords

Server-Side Registration Validation

```
1. // Validation middleware
2. function validateRegistration(req, res, next) {
3.   const { email, password } = req.body;
4.   if (!email || !password) {
5.     return res.status(400).json({ error: 'Email and password are required' });
6.   }
7.   if (!validator.isEmail(email)) {
8.     return res.status(400).json({ error: 'Invalid email format' });
9.   }
10.  if (password.length < 8) {
11.    return res.status(400).json({ error: 'Password must be at least 8 characters' });
12.  }
13.  if (!/[A-Z]/.test(password)) {
14.    return res.status(400).json({ error: 'Password must contain at least one uppercase letter'
15.  });
16.  }
17.  if (!/[a-z]/.test(password)) {
18.    return res.status(400).json({ error: 'Password must contain at least one lowercase letter'
19.  });
20.  }
21.  if (!/[0-9]/.test(password)) {
22.    return res.status(400).json({ error: 'Password must contain at least one number' });
23.  }
24.  if (!/^[A-Za-z0-9]/.test(password)) {
25.    return res.status(400).json({ error: 'Password must contain at least one special character'
26.  });
27.  }
28.  next();
29. }
```

```
24.   }
25.   next();
26. }
27.
```

(register.js)

This implements defense-in-depth by:

- Re-validating all inputs on the server side
- Using the validator library to perform email validation
- Checking each password requirement independently
- Providing specific error messages for security policy enforcement
- Following the principle that client-side validation is for UX, server-side validation is for security

Rate Limiting Protection

```
1. // Rate limiting middleware
2. const registrationAttempts = new Map();
3. function rateLimit(req, res, next) {
4.   const ip = req.ip || req.connection.remoteAddress;
5.   const now = Date.now();
6.   const attempts = registrationAttempts.get(ip) || [];
7.   const recentAttempts = attempts.filter(time => now - time < 3600000);
8.   if (recentAttempts.length >= 5) {
9.     return res.status(429).json({
10.       error: 'Too many registration attempts. Please try again later.'
11.     });
12.   }
13.   recentAttempts.push(now);
14.   registrationAttempts.set(ip, recentAttempts);
15.   next();
16. }
17.
```

(register.js)

This implements anti-automation protections by:

- Tracking registration attempts by IP address
- Limiting to 5 attempts per hour (3,600,000 ms)
- Returning a 429 (Too Many Requests) status code when limits are exceeded

- Protecting against registration brute force and enumeration attacks
- Implementing CIS Control guidance on limiting failed login attempts

Secure Password Storage


```
1. // Hash the password
2. const hashedPassword = await bcrypt.hash(password, saltRounds);
3.
4. // Insert user into database
5. const insertQuery = `
6.   INSERT INTO dbo.users (
7.     email,
8.     password,
9.     Role,
10.    status,
11.    registration_complete
12.  )
13.  VALUES (
14.    @email,
15.    @passwordHash,
16.    'user',
17.    'Active',
18.    1
19.  );
20.  SELECT SCOPE_IDENTITY() AS newId;
21. `;
22.
```

(register.js)

This implements secure credential storage by:

- Using bcrypt, a cryptographic hashing function designed for passwords
- Applying salt rounds to protect against rainbow table attacks
- Storing only the hash, never the plaintext password
- Using parameterized SQL queries to prevent SQL injection
- Assigning least-privilege role ("user") by default
- Following NIST 800-63B guidelines for secure password storage

2 Factor Authentication



Two-Factor Authentication

Enhance Your Account Security

Two-factor authentication adds an extra layer of security to your account by requiring both your password and a verification code from your mobile device.


Step 1: Download an Authenticator App

If you don't already have one, download an authenticator app on your mobile device:

[Google Authenticator](#)[Microsoft Authenticator](#)[Authy](#)

Step 2: Scan QR Code

Open your authenticator app and scan this QR code:




Can't scan? Use this secret key instead:

JVMDQPCXKAZVEZBVKJOSIORXJMYDQJSU

Step 3: Verify Setup


Enter the 6-digit verification code from your authenticator app:

Verify and Activate



Two-Factor Authentication

Enhance Your Account Security



Two-Factor Authentication Activated

Your account is now protected with 2FA. You'll need to enter a verification code from your authenticator app when you log in.

Recovery Codes

Save these recovery codes in a secure place. If you lose access to your authenticator app, you can use one of these codes to log in.

5D72-D377-EE38

00E3-6021-3BB5

0EB6-5386-0AF4

Each code can only be used once. Store them securely.

Proceed to Login

The 2FA setup page implements NIST SP 800-63B compliant multi-factor authentication by combining password authentication with a time-based token. Key security features include:

- TOTP Implementation:

- Uses industry-standard Time-based One-Time Password algorithm (RFC 6238) for generating secure, time-limited verification codes.
- Secure Secret Distribution:
 - Offers two secure methods for secret key distribution:
 - QR code for contactless, error-free setup
 - Base32-encoded text secret as accessibility fallback
- Authenticator App Integration:
 - Supports multiple standards-compliant authenticator apps (Google, Microsoft, Authy) for maximum compatibility.
- Verification Enforcement:
 - Requires immediate code verification before activation to prevent misconfiguration and account lockouts.
- Recovery Mechanism:
 - Implements secure recovery codes to prevent permanent loss of access.
- Clear User Guidance:
 - Three-step process with explicit instructions minimizes setup errors:
 - Download an authenticator app
 - Scan QR code/enter secret
 - Verify with a generated code

Code Explanation

2FA Setup Screen

```
1. // Frontend QR Code Generation and Display
2. async function setup2FA() {
3.   try {
4.     const response = await fetch('/api/2fa/setup', {
5.       method: 'POST',
6.       headers: {
7.         'Content-Type': 'application/json'
8.       },
9.       body: JSON.stringify({ userId: finalUserId, email: finalEmail })
10.    });
11.
12.    if (!response.ok) {
13.      const error = await response.json();
14.      throw new Error(error.error || 'Failed to set up 2FA');
15.    }
16.
17.    const data = await response.json();
18.
19.    // Display QR code and secret key
20.    qrcodeImage.src = data.qrCodeUrl;
21.    qrcodeImage.style.display = 'block';
22.    secretKey.textContent = data.secret;
23.
24.    // Store secret for verification
25.    window.tempSecret = data.secret;
26.  } catch (error) {
27.    showMessage(error.message || 'An error occurred during 2FA setup', 'error');
28.  }
29. }
30.
```

(2fa.html)

- Creates a temporary secret tied to a specific user
- Displays QR code for secure secret transmission
- Provides a fallback text secret for accessibility
- Error handling with user feedback for failed setup attempts

Verification Process

```

1. // Verify and activate 2FA
2. async function verify2FA() {
3.   const token = verificationCode.value.trim();
4.
5.   if (!token || token.length !== 6 || !/^\d+$/.test(token)) {
6.     showMessage('Please enter a valid 6-digit verification code', 'error');
7.     return;
8.   }
9.
10.  try {
11.    verifyButton.disabled = true;
12.    verifyButton.textContent = 'Verifying...';
13.
14.    const response = await fetch('/api/2fa/verify', {
15.      method: 'POST',
16.      headers: {
17.        'Content-Type': 'application/json'
18.      },
19.      body: JSON.stringify({ userId: finalUserId, token })
20.    });
21.
22.    if (!response.ok) {
23.      const error = await response.json();
24.      throw new Error(error.error || 'Failed to verify 2FA token');
25.    }
26.
27.    const data = await response.json();
28.
29.    // Display success screen with recovery codes
30.    setupContainer.style.display = 'none';
31.    successContainer.style.display = 'block';
32.
33.    // Display recovery codes
34.    if (data.recoveryCodes && data.recoveryCodes.length > 0) {
35.      recoveryCodesList.innerHTML = '';
36.      data.recoveryCodes.forEach(code => {
37.        const codeElement = document.createElement('code');
38.        codeElement.textContent = code;
39.        recoveryCodesList.appendChild(codeElement);
40.      });
41.    }
42.  } catch (error) {
43.    showMessage(error.message || 'Failed to verify the code', 'error');
44.    verifyButton.disabled = false;
45.    verifyButton.textContent = 'Verify and Activate';
46.  }
47. }
48.

```

(2fa.html)

- Validates 6-digit code format before submission
- Implements proper error handling and user feedback
- Disables button during verification to prevent multiple submissions
- Transitions to success screen only after server-side verification

Successful Activation Screen

```

1. // Backend code for generating recovery codes
2. const recoveryCodes = Array(3).fill(0).map(() => {
3.   const code = crypto.randomBytes(12).toString('hex');
4.   return `${code.slice(0,4)}-${code.slice(4,8)}-${code.slice(8,12)}`.toUpperCase();
5. });
6.
7. // Hash codes before storage
8. const hashedCodes = recoveryCodes.map(code =>
9.   crypto.createHash('sha256').update(code).digest('hex')
10. );
11.
12. // Save hashed codes to database
13. await pool.request()
14.   .input('userId', sql.Int, userId)
15.   .input('secretToStore', sql.NVarChar, user.twoFactorTempSecret)
16.   .input('recoveryCodesJson', sql.NVarChar, JSON.stringify(hashedCodes))
17.   .query(`
18.     UPDATE dbo.users
19.     SET
20.       twoFactorEnabled = 1,
21.       twoFactorSecret = @secretToStore,
22.       twoFactorTempSecret = NULL,
23.       twoFactorRecoveryCodes = @recoveryCodesJson
24.     WHERE id = @userId
25.   `);
26.

```

(2fa.js)

- Generates cryptographically strong recovery codes (36 bytes of entropy per code)
- Hashes recovery codes in the database for secure storage
- Clears the temporary secret after successful verification
- Activates 2FA flag in user record

Recovery Code Display

```

1. <div class="recovery-codes">
2.   <h4>Recovery Codes</h4>
3.   <p>Save these recovery codes in a secure place. If you lose access to your authenticator app,
4.   you can use one of these codes to log in.</p>
5.   <div class="code-list" id="recovery-codes-list">
6.     <!-- Recovery codes are inserted here -->
7.   </div>
8.   <p class="warning">Each code can only be used once. Store them securely.</p>
9. </div>

```

(2fa.html)

- Clear instructions for recovery code usage and storage
- One-time use policy for recovery codes
- Security warnings about proper code storage
- Separate display area with distinct styling for emphasis

File UploadThe image shows a web interface for secure document uploads. At the top, there is a yellow folder icon followed by the title "Secure Document Upload" in a large, bold, black font. Below this, a light gray rounded rectangle contains the heading "Upload Guidelines" in bold black text. Under the heading, four lines of text provide instructions: "Maximum file size: 10MB", "Allowed file types: PDF, DOC, DOCX, TXT, JPG, PNG, GIF, CSV, JSON, XML", "Files are scanned for security threats", and "Do not upload sensitive or confidential information". Below the guidelines box, there are three blue buttons with white text: "Select File", "Upload", and "← Back". Between the "Select File" and "Upload" buttons, the text "No file selected" is displayed in a smaller, italicized font.

This page allows authenticated users to upload documents to Azure Blob Storage securely. It is designed to:

- Limit accepted file types
- Restrict file size to 10MB
- Prevent unauthorized uploads
- Protect stored files using time-limited, permission-scoped Shared Access Signatures (SAS)

Code Explanation

```
1. const {
2.   BlobServiceClient,
3.   StorageSharedKeyCredential,
4.   generateBlobSASQueryParameters,
5.   BlobSASPermissions
6. } = require("@azure/storage-blob");
7. const { extractUserInfo } = require('./auth-utilities');
8.
```

(getSasToken.js)

These libraries are used to:

- Authenticate with Azure Blob Storage
- Generate SAS URLs (restricted upload tokens)

Handle preflight (OPTIONS) requests for CORS

```
1. res.setHeader('Access-Control-Allow-Origin', '*');
2. res.setHeader('Access-Control-Allow-Methods', 'GET, OPTIONS');
3. res.setHeader('Access-Control-Allow-Headers', 'Content-Type, Authorization');
4.
5. if (req.method === 'OPTIONS') {
6.   return res.status(200).end();
7. }
8.
```

(getSasToken.js)

This ensures the browser allows the backend code to interact with the API securely and correctly.

Validate incoming parameters

```
1. const blobName = req.query.blobName;
2. if (!blobName || typeof blobName !== 'string' || blobName.length > 256) {
3.   return res.status(400).json({ error: "Invalid blob name" });
4. }
5.
```

(getSasToken.js)

This prevents malicious input, such as trying to overwrite system files or abuse the storage endpoint.

SAS Token Generation

```
1. const expiresOn = new Date(new Date().valueOf() + 3600 * 1000); // 1 hour expiry
```

```
2.
3. const sasToken = generateBlobSASQueryParameters({
4.   containerName: containerName,
5.   blobName: blobName,
6.   permissions: BlobSASPermissions.parse("c"), // create permission
7.   expiresOn
8. }, sharedKeyCredential).toString();
9.
```

(getSasToken.js)

This creates a temporary, limited-use upload URL:

- Valid for 1 hour
- Only allows creating blobs (no reading or deleting)
- Specific to the file name given

Return the upload URL to the client

```
1. const sasUrl =
  `https://${accountName}.blob.core.windows.net/${containerName}/${encodeURIComponent(safeFileName)}?${sasToken}`;
2. return res.status(200).json({
3.
```

(getSasToken.js)

The frontend will then use this sasUrl to push the file directly to Azure Blob Storage securely, without exposing storage keys.

DOM Initialization

```
1. document.addEventListener('DOMContentLoaded', function() {
2.   const fileInput = document.getElementById('file');
3.   const filenameDisplay = document.getElementById('filename-display');
4.   const fileDetails = document.getElementById('fileDetails');
5.   const fileSize = document.getElementById('fileSize');
6.   const fileType = document.getElementById('fileType');
7.   const uploadButton = document.getElementById('uploadButton');
8.   const uploadMessage = document.getElementById('uploadMessage');
9.   // ...
10. });
11.
```

(upload.js)

This sets up references to all elements needed for upload: file input, buttons, progress bar, and messages.

File Selection & Display

```
1. fileInput.addEventListener('change', function() {
2.   if (this.files.length === 0) {
3.     filenameDisplay.textContent = 'No file selected';
4.     fileDetails.style.display = 'none';
5.     return;
6.   }
7.
8.   const file = this.files[0];
9.   filenameDisplay.textContent = file.name;
10.  fileSize.textContent = `${(file.size / 1024 / 1024).toFixed(2)} MB`;
11.  fileType.textContent = file.type;
12.  fileDetails.style.display = 'block';
13. });
14.
```

(upload.js)

When a file is selected, the script:

- Shows its name, size, and MIME type
- Hides this info if no file is selected
- Prevents uploads of undefined or zero-length files

When the user clicks Upload, the file is:

- Validated
- A secure upload URL is requested from the backend
- The file is uploaded directly to Azure using fetch() with a PUT request

This approach:

- Keeps credentials secure (client never sees storage keys)
- Prevents tampering (SAS URL is scoped and time-limited)
- Allows uploading files up to 10MB (as configured in the UI and possibly validated in backend)

Security Considerations

- Only the selected file is uploaded, scoped by filename and session
- No sensitive data is stored client-side (JWT and role-based access are handled elsewhere)
- Uploads are made directly to Azure Blob Storage, not through the app server, which improves performance and isolation
- Paired with `getSasToken.js`, this design implements the Principle of Least Privilege, ensuring that upload permissions are granted only temporarily and for specific actions

Block Dangerous File Extensions

```
1. // Block script files by extension
2. const fileExtension = path.extname(blobName).toLowerCase();
3. const blockedExtensions = [
4.   '.js', '.jsx', '.ts', '.tsx', '.php', '.asp', '.aspx',
5.   '.cgi', '.pl', '.py', '.sh', '.bat', '.cmd', '.ps1',
6.   '.vbs', '.vbe', '.jsp', '.html', '.htm', '.exe'
7. ];
8.
9. if (blockedExtensions.includes(fileExtension)) {
10.   console.log(`Blocked upload of script file: ${blobName}`);
11.   return res.status(403).json({ error: "Script files are not allowed" });
12. }
13.
```

(`getSasToken.js`)

This server-side check blocks files based on their extension to prevent the upload of potentially malicious scripts or executables. It supports secure input validation and system hardening.

Block Dangerous MIME Types

```
1. // Block script content types
2. const contentType = req.query.contentType;
3. const blockedContentTypes = [
4.   'application/javascript',
5.   'text/javascript',
6.   'application/x-javascript',
7.   'text/html',
8.   'application/xhtml+xml',
9.   'text/php',
10.  'application/x-httpd-php'
11. ];
12.
13. if (contentType && blockedContentTypes.includes(contentType)) {
14.   console.log(`Blocked upload with script content type: ${contentType}`);
15.   return res.status(403).json({ error: "Script content types are not allowed" });
16. }
17.
```

(getSasToken.js)

In addition to checking extensions, this snippet blocks dangerous MIME types, offering layered protection against browser-executable content like HTML or JavaScript files.

Filename Sanitization

```
1. // Helper function to sanitize filenames
2. function sanitizeFileName(filename) {
3.   return filename
4.     .replace(/[\/\?%*:|"<>]/g, '-') // Replace illegal characters
5.     .replace(/\.\./g, '-')           // Prevent directory traversal
6.     .replace(/^\./, '-')             // Prevent hidden files
7.     .trim();
8. }
9.
```

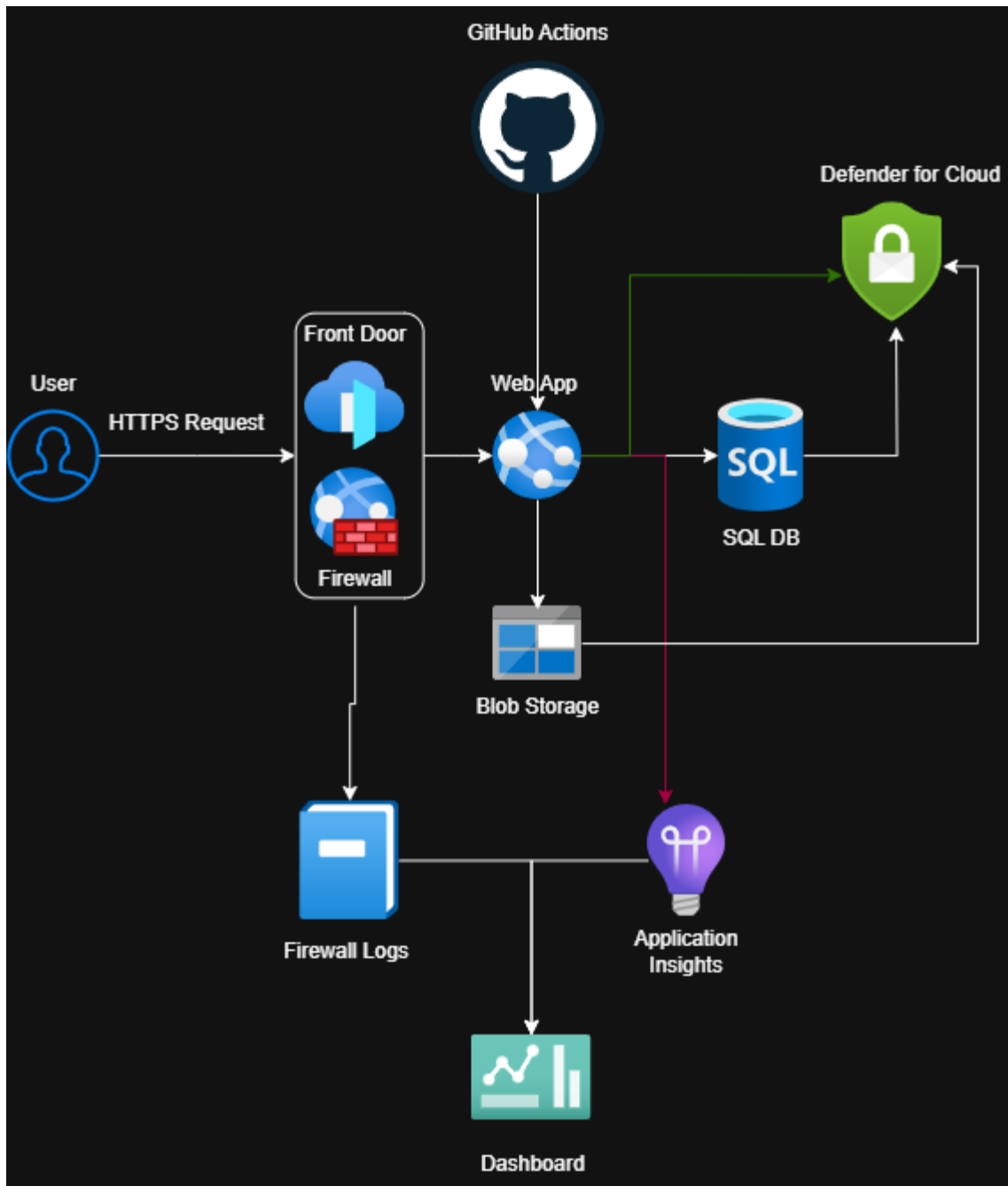
(getSasToken.js)

This function ensures uploaded filenames are cleaned of any illegal or dangerous characters. It prevents path traversal and file system abuse.

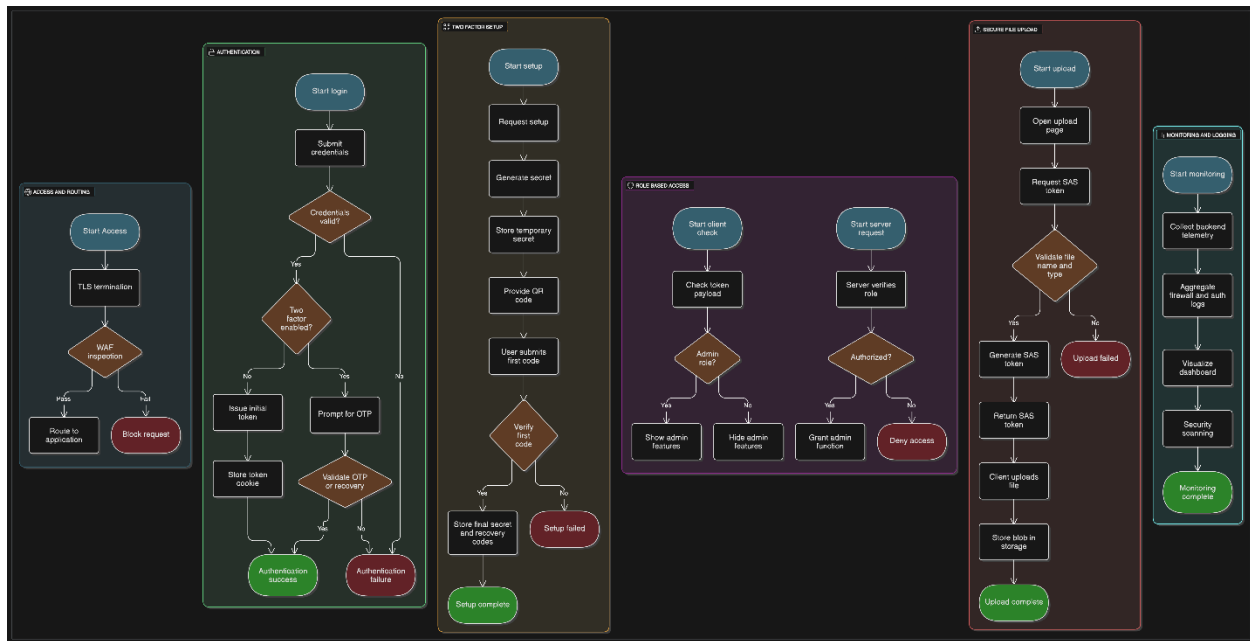
Proposed vs Final Architecture

Component	Proposed Architecture (based on diagrams)	Final Implementation	Notes / Reasoning
Frontend Hosting	Azure Static Web Apps	Azure App Service (Linux)	Static Web Apps replaced with App Service to support full backend API, TOTP authentication, and environment variable control
Authentication	Azure Active Directory B2C (AAD B2C)	Custom 2FA using TOTP, bcrypt, JWT	B2C dropped for simplicity and flexibility. Custom solution met MFA and session protection goals
Secrets Management	Azure Key Vault	App Service environment variables	Vault not used; secrets passed via env vars in App Service. Acceptable for non-prod. Vault recommended for scaling/rotation
Traffic Entry	Azure Front Door with custom WAF rules	Implemented as proposed	Front Door used with 20+ WAF rules (SQLi, XSS, SSRF, command injection, rate limiting), tested via curl
DDoS Protection	Azure Front Door & Platform-layer DDoS	In place implicitly	Azure handles DDoS automatically at the platform level, not manually configured
Encryption In Transit	HTTPS (TLS) enforced end-to-end	Fully implemented	TLS via HTTPS is used from the client to App Service to SQL over port 1433
Encryption At Rest	Azure Disk Encryption (BitLocker)	Transparent Data Encryption (TDE) on SQL	Azure SQL uses TDE by default; BitLocker is not relevant to App Service or SQL PaaS
Backend Hosting	Azure SQL in private subnet behind NSG	Azure SQL with firewall rules and TDE	VNet skipped; database isolated via IP rules and Azure Defender

Network Segmentation	NSG-Frontend / NSG-Backend	Not applicable	NSGs are not used due to the App Service deployment model. WAF used as the outer perimeter
Monitoring	Azure Monitor, Log Analytics, Defender for Cloud	Fully implemented	Dashboards include WAF rules hit count, request failures, and live metrics. Defender is active for all resources
Admin Access	Azure Cloud Shell / Backend VM access	Azure Cloud Shell used; no VMs created	Backend VMs replaced by PaaS components. Cloud Shell is used for CLI deployment and testing
DevSecOps / CI/CD	GitHub Actions + CodeQL	Implemented	GitHub Actions handles secure deployments; CodeQL is used for static analysis
Dynamic Testing	Not initially defined	OWASP ZAP DAST performed	ZAP scan revealed and verified CSP, cookie flags, and cache policy issues
Incident Response	Microsoft Sentinel	Implemented later for technical research purposes (publication)	Defender for Cloud covered alerts and visibility; Sentinel omitted due to scope



Network Diagram of the final architecture on Azure



This diagram visualizes the logical flow of user actions and system responses within the secure web application. It includes the TLS-enforced entry process, WAF inspection, login, and TOTP-based 2FA authentication, secure JWT issuance, and role-based access control for admin features. The file upload process is shown end-to-end, from SAS token generation to file validation and blob storage. The final section outlines the telemetry pipeline, including log aggregation, dashboard visualization, and real-time threat detection. This breakdown provides a detailed understanding of how application logic, security controls, and monitoring systems interact during runtime.

Hacking Week Evaluation

During Hacking Week, my secure web application (System Under Test) was targeted by red teamers simulating real-world attack techniques. Their goal was to identify weaknesses in authentication, access control, file handling, and server-side protection.

Based on the test results documented in their report, several key vulnerabilities were identified:

1. Client-Side Weaknesses

- The application relied on localStorage for token handling in the early design phase. This exposed JWTs to potential XSS attacks.

2. Incomplete 2FA Protection

- The red team discovered that the 2FA verification endpoint lacked proper rate limiting and allowed direct access without prerequisite authentication steps.
- Their attack simulated brute forcing 6-digit OTPs and demonstrated user enumeration through error messages.

3. File Upload Vulnerabilities

- The upload endpoint was bypassed using tools like curl, where attackers modified file extensions (e.g., .php.jpg) and Content-Type headers to submit potentially dangerous files.
- No server-side content inspection (e.g., magic byte analysis) was in place at the time of testing.

4. WAF Configuration Gaps

- Azure Front Door's WAF was found to be running in detection mode instead of prevention mode.
- Special characters and wildcard inputs passed through filters unblocked, reducing the effectiveness of edge-layer defenses.

5. Rate Limiting Bypasses

- Attackers evaded IP-based rate limits using VPNs or anonymized traffic (e.g., TOR), which the WAF did not detect.
- No user fingerprinting or behavioral rate tracking was in place.

Reflection

This project allowed me to move from theory to practice in the field of secure web development and cloud architecture. At the start, I had limited experience with Azure services and professional-level security controls. However, through research, implementation, and testing, I was able to build and secure a full web application in the cloud.

During the process, I learned how important it is to apply security at every layer of a system. I implemented multi-factor authentication (2FA), role-based access control, secure file uploads using SAS tokens, and encrypted database communication. I also learned to monitor and analyze system activity using tools like Application Insights, Log Analytics, and Defender for Cloud.

One of the most valuable parts of the project was Hacking Week. This experience gave me direct feedback on my system's strengths and weaknesses. The red team was able to find and exploit some gaps in my implementation, such as weak 2FA protection logic and the use of client-side token checks.

From this experience, I now understand that building a secure system is not just about following best practices but also requires testing, reviewing, and learning from real-world feedback. I also learned that proper security includes not only technical controls, but also clear logging, visibility, and recovery processes.

Overall, I am proud of the secure environment I built and the knowledge I gained throughout the project. I now feel more confident in applying secure development practices and evaluating the risks and

protections needed in cloud-based systems. This project has prepared me well for future work in cybersecurity and cloud infrastructure.

Addressing Research Questions

Main Research Question:

How can a secure web application be designed and deployed on Azure in a way that meets professional security and compliance requirements defined by CIS and NIST standards?

Through a combination of layered security controls, Azure-native services, and secure coding practices, I deployed a compliant web application that includes MFA, encrypted communication, secure storage, monitoring, and threat detection. Each control was mapped to CIS Controls and NIST SP 800-53 categories, including authentication, input validation, secure storage, and audit logging. The system was validated through DAST, CodeQL static scans, and red team testing during Hacking Week.

Sub-Questions:

Question	Answer
<i>Which cloud-native threats pose the most significant risk to web applications, and how do professionals mitigate these?</i>	Threats like SQL injection, XSS, brute force, misconfigured storage, and insecure APIs were addressed using WAF rules, token-based access, rate limiting, and role-based controls.
<i>Which architectural, configuration, and implementation decisions most significantly affect the security, maintainability, and scalability of the deployed solution?</i>	Using Azure Front Door with WAF, role-based APIs, GitHub Actions CI/CD, and SAS token-based uploads ensured maintainability while securing each data path.
<i>How can the final implementation be evaluated to demonstrate alignment with professional standards and compliance frameworks?</i>	Evaluations included WAF log analysis, GitHub CodeQL output, MITRE alignment, and post-Hacking Week adjustments to fill gaps (e.g., 2FA logic improvements).

BoK Checklist

Must Have

Topic	Status	Evidence
Standards and Regulations	Completed	Mapped controls to CIS/NIST, including OWASP Top 10
Threat Analysis & Secure Design	Completed	STRIDE threat model, misuse cases, Zero Trust principles
Authentication with MFA	Completed	TOTP-based 2FA with bcrypt + recovery codes
Encryption (Data in Transit & Storage)	Completed	HTTPS enforced, TLS to SQL, TDE, secure cookie + SAS token use

Additional Topics

Topic	Status	Evidence
Principle of Least Privilege	Completed	Role-based access (JWT), restricted endpoints, SQL access via Managed Identity
Input Validation	Completed	Client + server validation, file type filters, SQL input checks
System Security and Hardening	Completed	Removed unnecessary headers, used CSP, disabled x-powered-by, secure deployment cleanup
Logging and Monitoring	Completed	Azure Monitor, Log Analytics, App Insights, Defender for Cloud dashboard
Cloud Security	Completed	Blob container access policy, firewalls, Front Door routing, managed identity
Secure Updates / CI/CD	Completed	GitHub Actions with CodeQL, clean release builds, secure deployment via OIDC
User Privacy & Data Protection	Completed	Hashed passwords and secrets, CSP headers, proper logout handling
DAST	Completed	OWASP ZAP scan on production, findings addressed.