Web Application Development Guide

Introduction

Below is a suggested high-level breakdown of the project into various parts, along with brief descriptions for each. The idea is to provide an "instruction manual" style outline, written in clear, beginner-friendly language. The goal is to help you understand the logical flow of what you need to do, from planning your web app to finally deploying it on Azure using the free student services.

1 Project Planning and Conceptual Design

Purpose

- Understand what the application should do (upload, store, and share images or videos, provide ratings and comments, handle user roles, etc.).
- Figure out how the different pieces (front-end website, REST API, database, storage, authentication) will work together.

Key Tasks

1. **Requirements Gathering**
   - Confirm functional requirements:
     - Creator users can upload videos/photos, add metadata.
     - Consumer users can view/search, comment, rate.
   - Confirm non-functional requirements:
     - Scalability, performance, security.
2. **Architectural Diagram**
   - Sketch a simple diagram of the system:
     - Web Front-End (HTML/JavaScript)
     - REST API (hosted on Azure App Service or similar)
     - Database (e.g., Azure SQL Database or Cosmos DB)
     - Storage (Azure Blob Storage for images/videos)

- Authentication/Authorization (Azure Active Directory B2C or another identity service)

3. **Determine Technology Choices**
   - Decide on front-end tech (raw HTML/JavaScript or a simple framework like React if comfortable).
   - Decide on back-end tech (a simple Node.js, Python Flask, or C# .NET API).
   - Decide on your database (SQL vs NoSQL) and storage (Blob Storage).

Outcome

A clear, written outline or document showing what the app will do and how each part will connect.

2 Setting Up Azure Resources

Purpose

Prepare your Azure environment so you can deploy all necessary components.

Key Tasks

1. **Create an Azure Account**
   - Sign up for an Azure for Students subscription using your student account (no credit card required).
2. **Organize Your Resources**
   - Create a Resource Group in Azure (think of this as a "folder" to hold everything for your project).
   - Decide on naming conventions so you can keep track of your resources easily.
3. **Azure Services to Enable**
   - **App Service** (to host your REST API, optionally also host the front-end if you prefer).
   - **Storage Account** with Blob Containers (to store media files).
   - **Azure SQL Database** or **Cosmos DB** (for user data, metadata, and comments).
   - **Azure Active Directory B2C** or other identity service (to manage user logins and roles).

Outcome

A properly configured Azure environment ready to host the different parts of your web application.

3 Database and Storage Setup

Purpose

Provide a place to store your application data (users, images/videos, comments, ratings, etc.) and a place to store large media files.

Key Tasks

1. **Choosing Your Database**
   - If you prefer a relational database: Use Azure SQL Database.
   - If you prefer a NoSQL approach: Use Azure Cosmos DB.
2. **Schema or Data Model**
   - Identify what "tables" or "collections" you will need:
     - **Users**: consumer vs creator roles.
     - **Media Items**: video/photo metadata (title, caption, location, etc.).
     - **Comments/Ratings**: referencing which media item they belong to.
3. **Set Up Storage Containers**
   - In Azure Storage Account, create a Blob Container (e.g., `media-uploads`) to hold your video or photo files.
   - Make sure you note the connection strings or keys to access Blob Storage from your API.

Outcome

A working database or storage container set up on Azure, ready for data insertion and retrieval.

4 Building the RESTful API

Purpose

Provide an interface for the front-end to interact with your database and storage. This API will handle all logic, such as creating user profiles, uploading images/videos, retrieving them, etc.

Key Tasks

1. **Set Up the Development Environment**
    - Decide on language/framework (for instance, Node.js with Express, Python with Flask, or C# with ASP.NET Core).
    - Initialize your project folder locally or in an online environment (e.g., GitHub Codespaces or Visual Studio Code).
2. **Design the Endpoints**
    - **Creator Endpoints**: e.g., `POST /api/media`, `PUT /api/media/{id}` to upload and edit metadata.
    - **Consumer Endpoints**: e.g., `GET /api/media`, `POST /api/media/{id}/comment`, etc.
3. **Database Integration**
    - Write code to connect to Azure SQL/Cosmos DB.
    - Implement data models or schemas.
    - Implement CRUD operations (Create, Read, Update, Delete) for your content.
4. **Blob Storage Integration**
    - Write code that handles file uploads to Azure Blob Storage.
    - Generate secure URLs for streaming or displaying images/videos.
5. **Logging and Error Handling**
    - Make sure you have basic error messages for debugging.
    - Return user-friendly error messages to the front-end.

Outcome

A functional REST API that can store, retrieve, and manage user data and media files.

5 Implementing Authentication and Authorization

Purpose

Ensure only the right people can access or modify certain features. For example, creators can upload, consumers can only view.

Key Tasks

1. **User Roles and Accounts**
   - Decide how you will mark a user as either "creator" or "consumer."
   - Create a column/field in your database for user role.
2. **Authentication Flow**
   - Use Azure Active Directory B2C or similar:
     - Integrate it with your API to verify tokens or use OAuth flows.
     - Decide how new consumer users sign up (maybe a sign-up link).
     - For creator users, you might manually create them or invite them.
3. **Secure Your Endpoints**
   - For instance, `POST /api/media` might only allow requests from users with the "creator" role.
   - Restrict other endpoints based on user roles.
4. **Testing**
   - Test sign-up, sign-in, and access flows.
   - Confirm that unauthorized attempts are blocked.

Outcome

A secure API enforcing role-based access, ensuring creators and consumers have correct permissions.

6 Building the Front-End (Consumer & Creator Views)

Purpose

Develop the user interfaces for both consumer and creator users, making it easy to interact with the back-end REST API.

Key Tasks

1. **Consumer View**
   - **Home/Search Page**: lists available media items, includes a search bar.
   - **Media Detail Page**: shows the photo/video, comments, and rating feature.

- **Comment/Rating Interface**: forms to submit comments or ratings.

2. **Creator View**
   - **Upload Page**: form to upload a new image/video, set title, caption, location, tags, etc.
   - **Manage Page**: list of previously uploaded items, with the option to edit metadata or remove.
3. **API Integration**
   - Use JavaScript (or a framework like React) to call the REST API endpoints.
   - Display the returned data (e.g., fetch the list of images or a single image with its comments).
4. **Basic Design**
   - Keep it simple. Basic HTML/CSS is enough.
   - If you want, use a minimal UI library or a pre-built CSS framework (like Bootstrap) to make it look decent.

Outcome

A working front-end that consumers and creators can use to interact with the service.

7 Hosting the Front-End (Static Web Hosting or Azure App Service)

Purpose

Make the front-end available online so users can access the website from a public URL.

Key Tasks

1. **Option A: Azure Static Web Apps**
   - Deploy your HTML/CSS/JavaScript to Azure Storage for static web hosting.
   - Connect it to your REST API URL so it can fetch data properly.
2. **Option B: Azure App Service**

- Host both the back-end and front-end on the same App Service or separate ones.
- Configure the server to serve the front-end files.

3. **DNS and Custom Domain (Optional)**
   - If you have a custom domain, configure Azure DNS or an external DNS service.
   - This part can be skipped if you're okay with a .azurewebsites.net or .web.core.windows.net URL.

Outcome

Your website's front-end is live and accessible from a public web address, ready for testing by real users.

8 Scalability

Purpose

Ensure your solution can handle increased traffic and remains reliable. Add extra features if resources allow.

Key Tasks

1. **Caching**
   - Implement caching for frequently accessed data (e.g., Azure Cache for Redis) to reduce database load.
2. **Media Conversion Services**
   - If you want to offer automated thumbnail creation or video transcoding, explore Azure Media Services.
   - Remember to check free tier limits.
3. **Monitoring & Logging**
   - Set up Azure Monitor or Application Insights to track performance, errors, usage stats.

Outcome

A more advanced, efficient application that can scale under higher traffic.

9 Testing and Final Demonstration

Purpose

Verify that each component works together properly and demonstrate to others that the project meets its objectives.

Key Tasks

1. **Functional Testing**
    - Walk through main tasks: sign up as consumer, sign in, search for images, comment/rate, sign in as creator, upload new content, etc.
2. **Performance Testing (basic)**
    - Test how many users can be served with the free tier.
    - Check if the site remains responsive with multiple concurrent requests.
3. **Final Presentation**
    - Provide a live URL, a quick demo, and a short explanation of how everything works together.

Outcome

A completed, deployed, and tested web application that fulfills the project's requirements.

Wrapping Up

These steps outline everything from planning and designing the system, to setting up Azure resources, building the database and REST API, implementing security, creating front-end views, deploying, and finally scaling and testing.