Software Design Document (SDD)

Team 3

NewsVerse

Software Engineering
Fall 24

CS 673 A1

Damanjit Singh
Abigail Gualda
Stephen Yang
Dewei Wang
Yuhang Wang

1. **Project and SDD Overview**

   Overview:

   NewsVerse is a news aggregation platform that leverages AI semantic analysis to present balanced news views from multiple sources. Designed with a user-centered approach, NewsVerse delivers personalized news feeds, real-time updates, and interactive features that promote user engagement and informed discussion. The platform's AI-driven analysis prioritizes diverse perspectives, making it a reliable source for well-rounded news. NewsVerse integrates seamlessly across devices, allowing users to access up-to-the-minute information on topics that matter most to them.

   Purpose:

   The purpose of this Software Design Document (SDD) is to provide a comprehensive outline of the NewsVerse platform's architecture and design choices. This document details the structural, functional, and interface components of the application, illustrating how NewsVerse's design supports its mission to deliver balanced, personalized news through advanced AI technologies. It serves as a guide for developers, stakeholders, and technical teams to ensure alignment in implementing and maintaining the platform's features, performance, and scalability goals

   Scope:

   NewsVerse aggregates and delivers news from multiple sources with a focus on providing balanced views through AI semantic analysis. It supports personalized news feeds, real-time updates, and interactive features to facilitate user engagement around the platform.

2. **System Overview:**

   NewsVerse aims to provide a balanced and personalized news experience by aggregating articles from multiple reputable sources and using AI-driven semantic analysis to highlight diverse perspectives on trending topics. The platform is designed to encourage informed engagement and provide users with up-to-date information on areas of interest.

   System Functionality:

   News Aggregation: Pulls news from various sources, ensuring a wide range of coverage and viewpoints.

   AI Semantic Analysis: Uses AI to analyze and present balanced perspectives, reducing bias and highlighting different angles on key topics.

Real-Time Updates: Provides live updates to keep users informed as stories evolve.

Interactive Features: Allows users to engage with content through comments, shares, and reactions, fostering community discussion.

High-Level Requirements:

Data Integration: Seamlessly collect, parse, and store data from multiple external news APIs or sources.

AI Processing: Apply AI algorithms for sentiment and semantic analysis to ensure balanced representation of news.

Scalability and Real-Time Functionality: Support high-volume traffic and provide near real-time updates for breaking news.

User Engagement Features: Implement interactive elements such as comments, reactions, and sharing options within the platform.

3. Diagrams

Decision Making Diagram Figure 1:

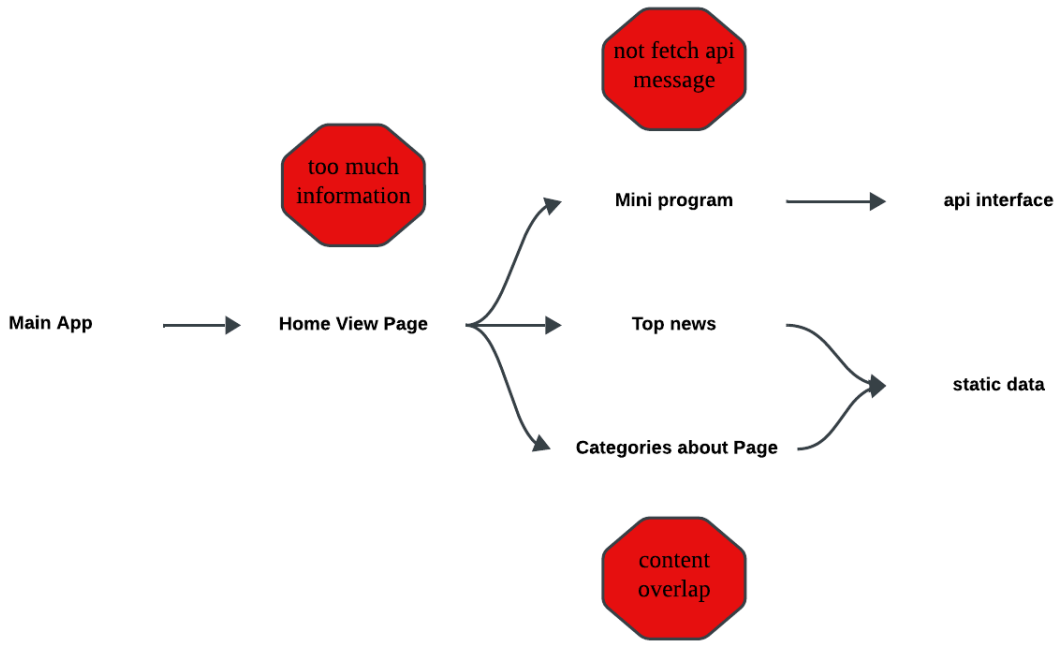

Figure 1 Description:

The diagram is a visual representation of the structure and flow of information within our website. The main components include the "Main App", which connects to the "Home View Page". From the Home View Page, there are several linked elements, such as a "Mini program", "Top news", and "Categories about Page". The diagram also highlights two potential issues with the design - "too much information" and "not Fetch api message", suggesting areas for improvement or optimization. Overall, the diagram depicts the high-level architecture and potential pain points of the application or website, which could be useful for understanding the software's structure and identifying areas for enhancement.

Reward Factors:

The serverless architecture provides key benefits, including enhanced scalability, as AWS Lambda and API Gateway automatically handle fluctuating workloads, accommodating real-time requests without manual intervention. Performance is optimized with AWS DynamoDB's fast, flexible NoSQL storage and AWS Batch for efficient batch processing. This design also aligns with future goals by allowing modular expansions, seamless integration with additional AWS services, and cost-effective scaling, ensuring the application is well-prepared for evolving user demands and growth.

4. Architectural Design

System Overview

1. Architecture Design of Software Diagram: It illustrates the front-end and back-end connection, AI-based semantic scoring / grouping, and database integration.

Architectural Design Figure 2:

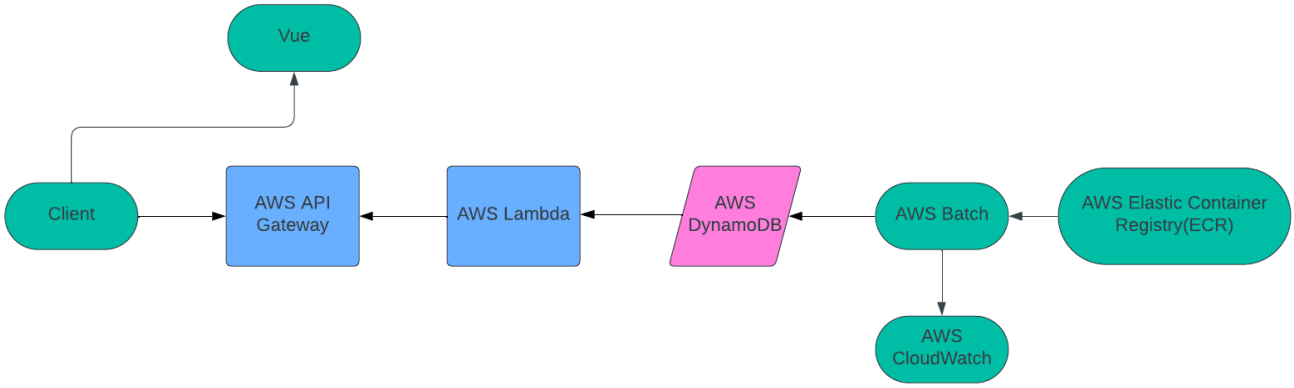

Figure 2 Description:

The architectural design illustrates a serverless application workflow using AWS services to handle client interactions, data processing, and storage. The client communicates with the system via a front-end built with Vue, which sends requests through AWS API Gateway. API Gateway routes these requests to AWS Lambda functions, which perform the necessary back-end operations. AWS DynamoDB serves as the data storage solution, providing a NoSQL database to store and retrieve information efficiently. For batch processing, AWS Batch retrieves data from DynamoDB, processing it in bulk as needed, with monitoring and logging managed by AWS CloudWatch. Additionally, AWS Elastic Container Registry (ECR) holds container images that AWS Batch uses, allowing for seamless scalability and efficient execution of batch jobs. This setup ensures a streamlined, serverless architecture capable of handling real-time requests, persistent data storage, and large-scale batch processing.

Data Flow Figure 3:

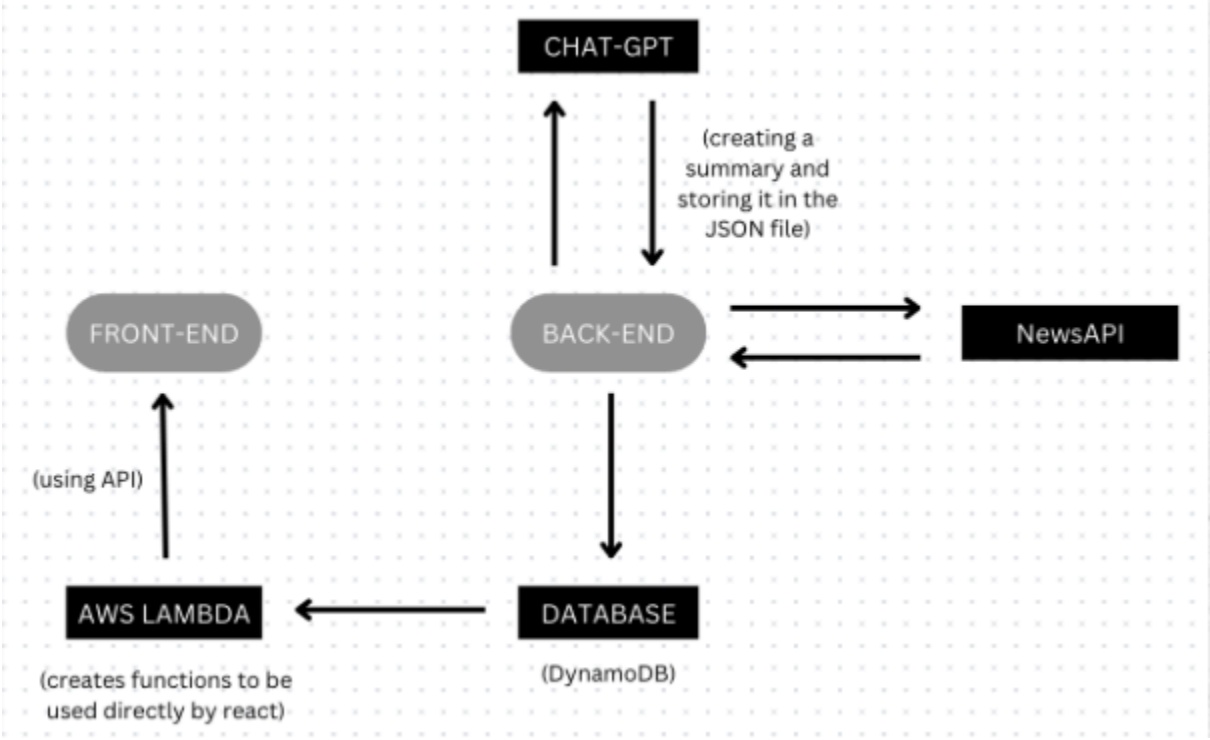

Figure 3 Description:

This diagram illustrates the architecture of a web application that utilizes ChatGPT, a language model developed by OpenAI, to provide enhanced functionality.

The architecture consists of three main components:

1. Front-end: This is the user-facing part of the application, which interacts with the ChatGPT model via an API. The front-end likely uses this API to gather information or generate content as needed.

2. Back-end: This component acts as an intermediary between the front-end and other backend systems. It interacts with ChatGPT to create summaries or store information in a JSON format. It also communicates with a database, likely DynamoDB, to persist data.

3. NewsAPI: This represents an external API that the application may use to fetch news or other information to be integrated into the user experience.

The diagram also shows that the back-end utilizes AWS Lambda, a serverless computing service, to create functions that can be directly accessed by the front-end. This suggests an event-driven, scalable architecture that can seamlessly handle user requests and interactions powered by the ChatGPT language model.

Overall, this diagram depicts a modern, cloud-based web application that leverages the capabilities of ChatGPT to enhance the user experience, with a clear separation of concerns between the front-end, back-end, and external data sources.

Sequence Diagrams Figure 4:

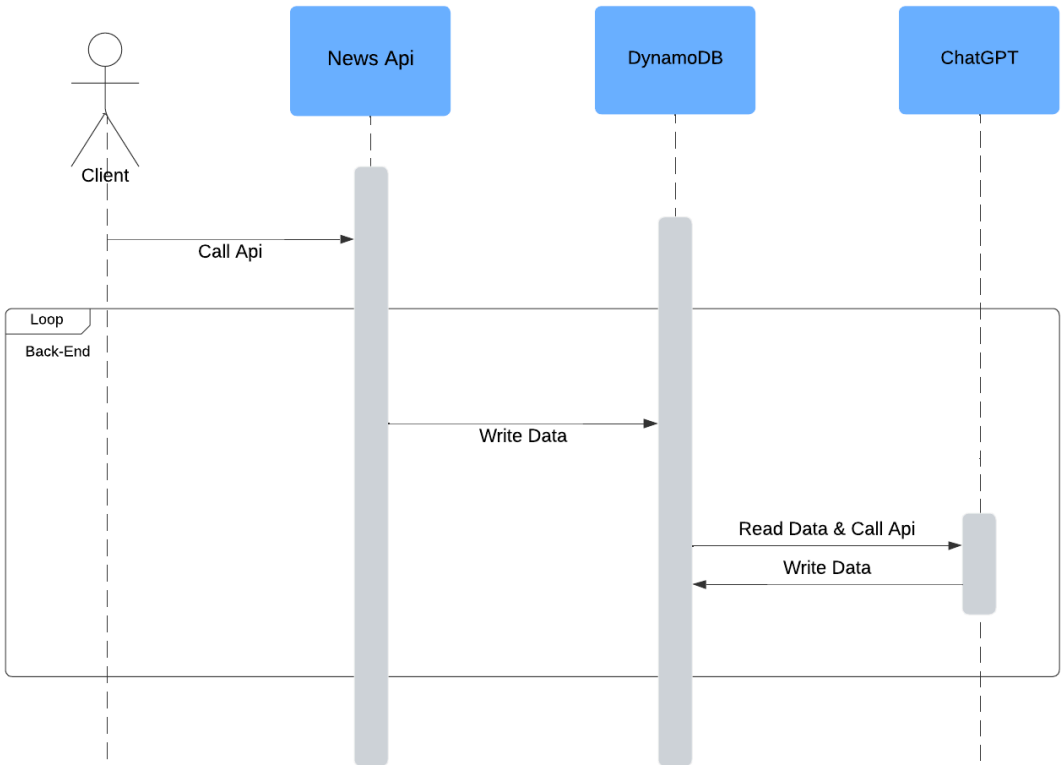

Figure 4 Description:

This is a sequence diagram illustrating a system architecture where a Client initiates a process by calling an API. The flow shows interactions between four components: the Client, a News API, DynamoDB (a database service), and ChatGPT. When the Client makes an API call, it triggers a loop in the back-end process where the News API writes data to DynamoDB, and then ChatGPT both reads data from DynamoDB and writes data back to it after processing. The news data is collected, stored, and then processed or analyzed using ChatGPT, with DynamoDB serving as the intermediate storage layer between these services.

5. Endpoint Details:

1) **Everything Endpoint**
   - URL: /v2/everything
   - Description: Searches articles from over 150,000 sources published in the last 5 years, ideal for news analysis and article discovery.

Response Schema:


```
{

  "status":"ok",
  "totalResults":7046,
  "articles":[
     {
        "source":{
           "id":"wired",
           "name":"Wired"
        },
        "author":"Joel Khalili",
        "title":"The World’s Biggest Bitcoin Mine Is Rattling This Texas Oil Town",
        "description":"A cash-strapped city in rural Texas will soon be home to the world’s largest bitcoin mine. Local protesters are “raising hell.”",

"url":"https://www.wired.com/story/the-worlds-biggest-bitcoin-mine-is-rattling-this-texas-oil-town/",

"urlToImage":"https://media.wired.com/photos/66c5ecc5724cee849e3104da/191:100/w_1280,c_limit/WIRED_BTC_EC_B-Elena-Chudoba.jpg",
        "publishedAt":"2024-09-11T10:00:00Z",
```

```
        "content":"The previous October, Sawicky organized a weeklong
protest alongside environmental activist group Greenpeace and brandished
various anti-bitcoin signs at anyone who entered the Riot facility. Only
a … [+3641 chars]"
      }
]
```


2) **Top headlines Endpoint**
   - **URL:** /v2/top-headlines
   - **Description:** Returns breaking news headlines based on country, category, or publisher, suitable for live news tickers.

6. Request Parameters:

- apiKey (required): Your API key (can also be sent via the X-Api-Key HTTP header).
- country: 2-letter ISO 3166-1 country code (e.g., us). Cannot be used with the sources parameter.
- category: Category for headlines (options: business, entertainment, general, health, science, sports, technology). Cannot be used with the sources parameter.
- sources: Comma-separated identifiers for specific news sources. Cannot be used with country or category.
- q: Keywords or phrases for search.
- pageSize: Number of results per page (default 20, max 100).
- page: For paginating through results if total results exceed the page size.

Backend Class Diagram Figure 7:

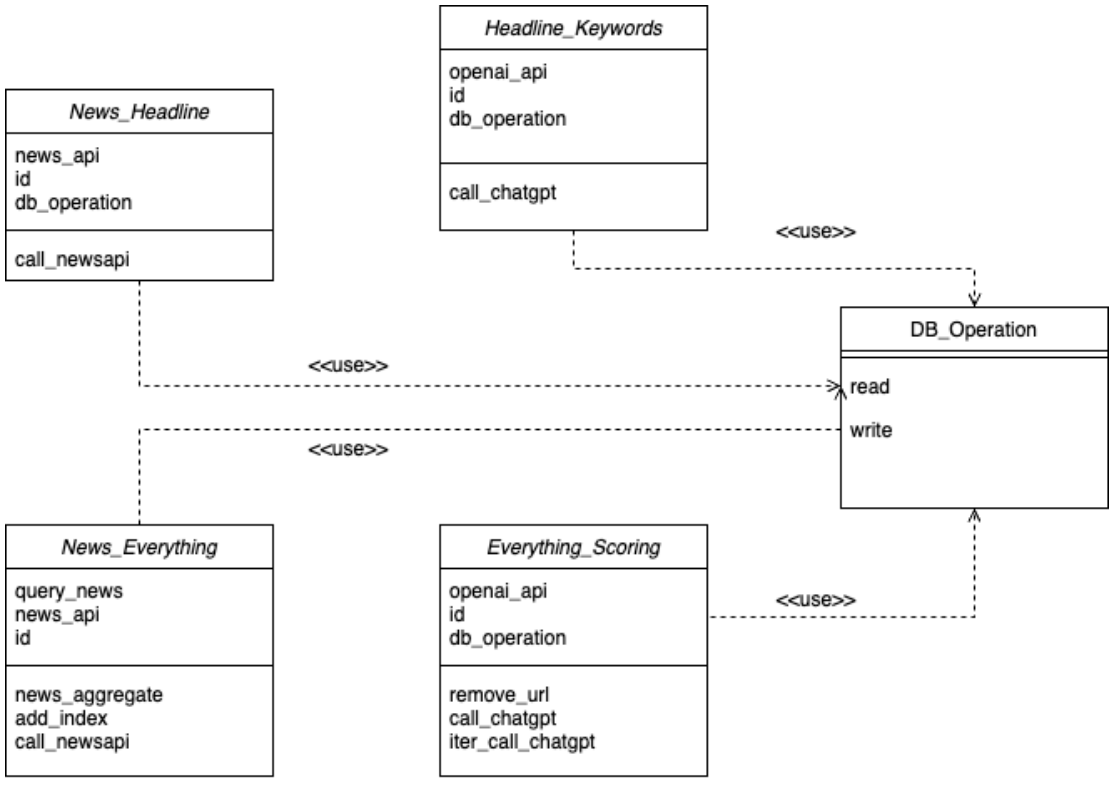

Figure 7 Description:

This diagram represents the backend class structure of an application that utilizes ChatGPT, a natural language processing model, to provide various functionalities.

The key components include:

1. News_Headline: This class likely handles the processing and retrieval of news headlines, with properties such as 'call_newsapi'.

2. News_Everything: This class appears to deal with more general news-related data and terms, also with a 'call_newsapi' property. In order to aggregate the news across categories, we use the news_aggregate function to organize the data associated with each category to its unique key in the dictionary, then store it in the database.

3. Everything_Scoring: This class seems to be responsible for scoring or evaluating 'bots_resource' data, potentially using the 'call_chatgpt' property to interact with the ChatGPT model. The iter_call_chatgpt function will put multiple versions of ChatGPT in place in the backend to ensure it will switch between the versions once the token of one has run out.

4. DB_Operation: This class likely encapsulates the database-related operations, such as reading and writing data, with 'read', and 'write' methods.

5. Headline_Keywords: This class appears to hold various metadata-like properties, such as 'id' is the index to write data to the database, 'openai_api', 'db_operation', and 'call_chatgpt', which could be used to enhance the processing of news headlines.

The diagram suggests a structured, object-oriented approach to the backend design, with clear separation of concerns and potential integration with ChatGPT to leverage its natural language processing capabilities within the application.

7. Backend Services

- News Aggregation: Collects news data from various APIs and stores it in DynamoDB.
- Keywords Generation: Fetch the news in DynamoDB and trigger the ChatGPT API to generate the keywords for the news then store it in DynamoDB.
- News Grouping and Semantic Analysis: Fetch the news in DynamoDB and trigger the ChatGPT API to provide the semantic scores and segmentation of the news then store it in DynamoDB.

Frontend Entity–relationship Model Diagram Figure 8:

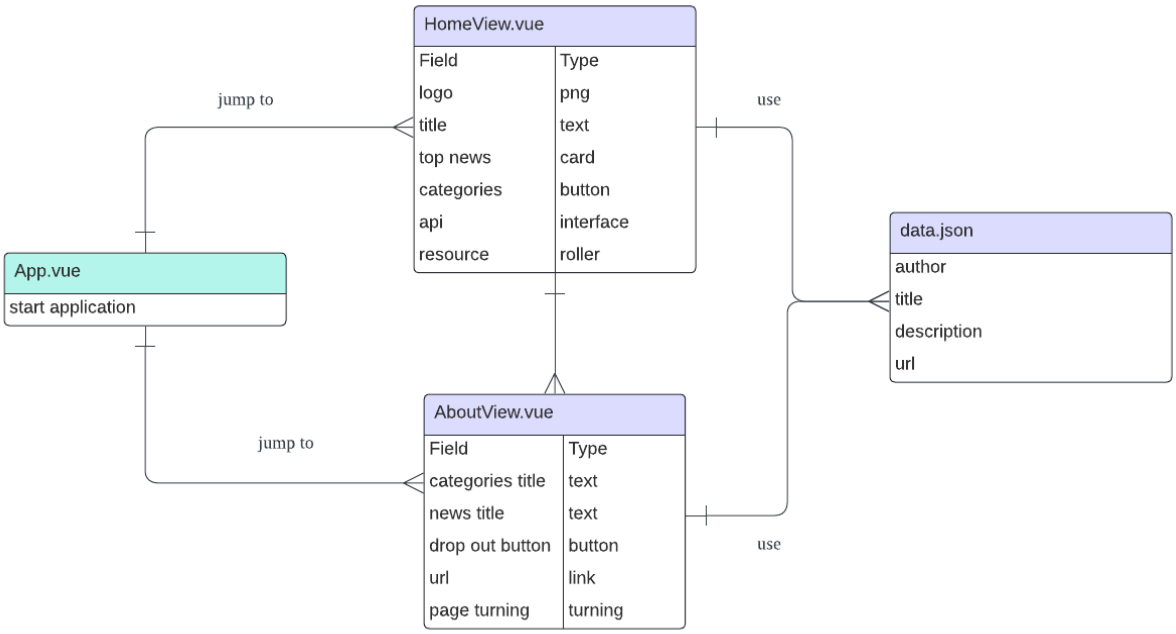

Figure 8 Description:

This diagram depicts the entity-relationship model for the frontend of an application. The main components are:

1. App.vue: This represents the main view of the application, with properties like "start_application" that likely initiate the application's functionality.

2. HomeView.vue: This view contains various elements, such as "logo", "top news", "categories", "api_interface_router", and "data_json". These elements likely make up the homepage of the application.

3. AboutView.vue: This view contains elements like "categories title", "news title", "drop out button", "url", and "page turning". These elements likely comprise the about or information page of the application.

4. Data.json: This represents the data model for the application, with properties like "author", "title", "description", and "url". This data is likely used to populate the content displayed in the various views.

The diagram suggests a modular and organized frontend design, with clear separation of concerns between the different views and their respective elements. This structure can help ensure maintainability and facilitate future development and expansion of the application.

8. Functional Description

User Interface Design Figure 9:

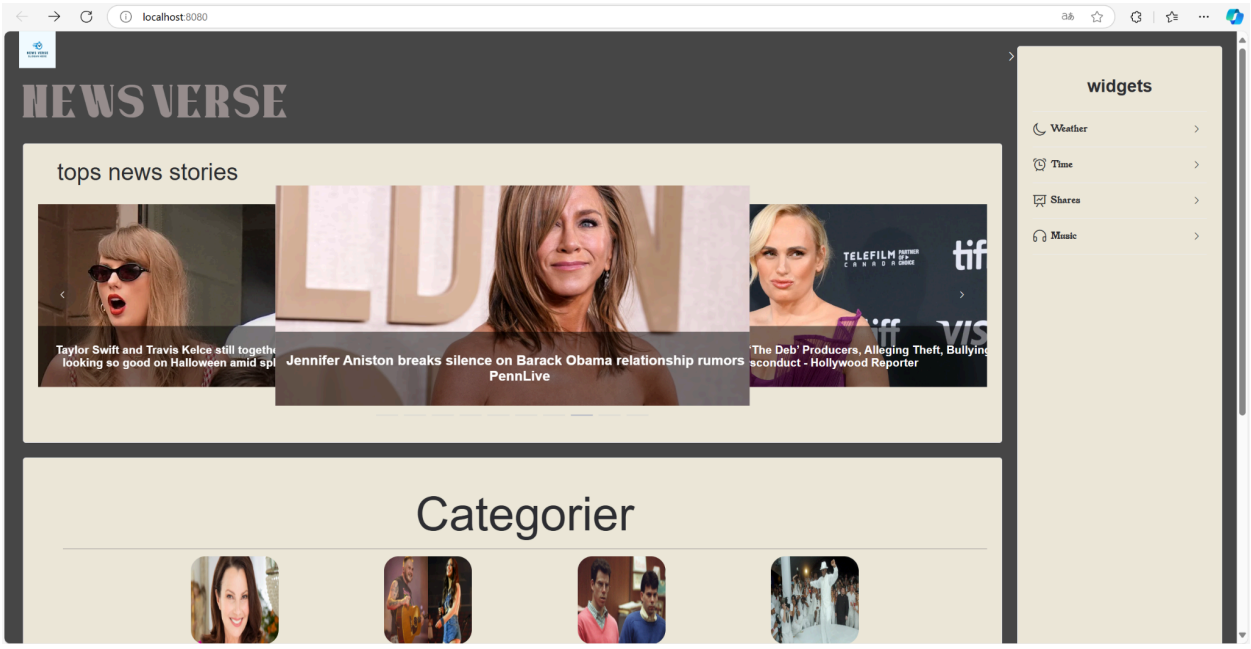

Figure 9 Description:
The above is our main landing page which has a top news column containing the current popular news.

User Interface Design Figure 10:

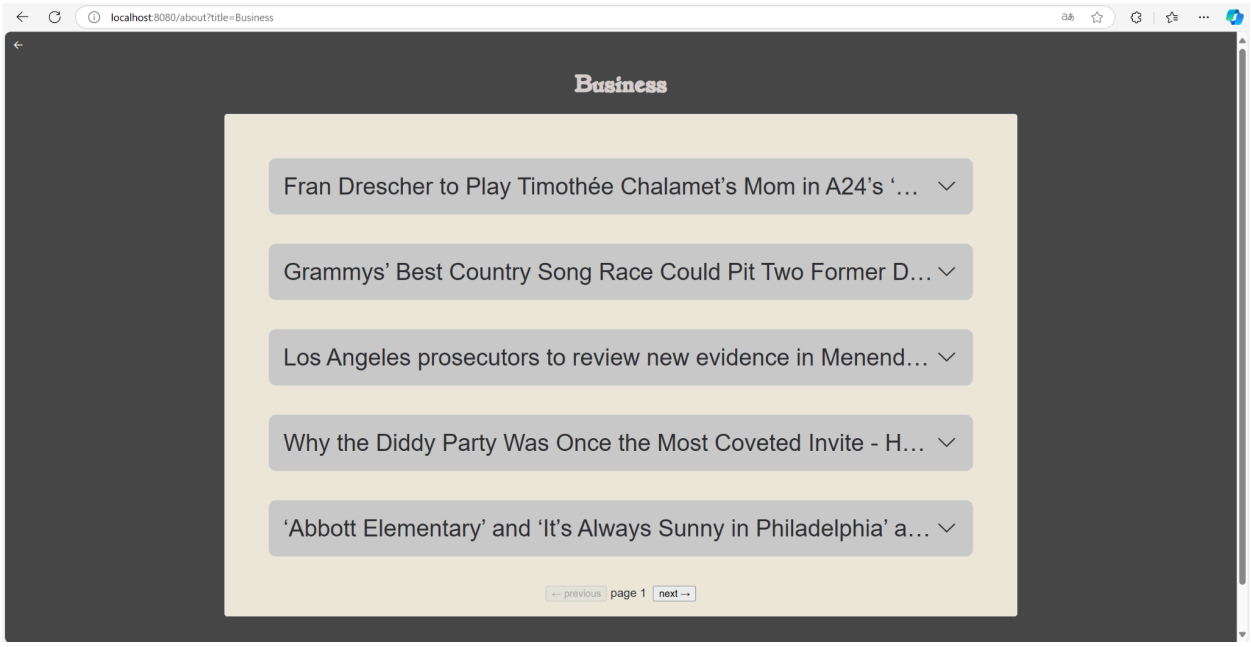

Figure 10 Description:

This is the screenshot of the landing page of the categorical news that a user can choose from.

Testing

1. Unit Testing:
   Focuses on testing individual components in the front-end, backend modules to ensure each component is out of bug.
2. Integration Testing:
   Verifies that the front-end components correctly display data received from the backend database. Ensure the integrated application runs properly from which the outputs are as expected overall.
3. Performance Testing:
   Load tests to ensure system stability under peak traffic.

9. Error Handling

The system will employ a structured error handling strategy to ensure reliability. Errors will be detected through status codes and exception handling mechanisms. Each error will be classified (e.g., client-side, server-side) and managed accordingly:

- **Client-side Errors**: Promptly notify users of invalid input or missing parameters, returning relevant error messages.
- **Server-side Errors**: Capture unexpected issues and retry failed processes where feasible. If unrecoverable, log details and notify the support team.
- **Reporting**: All critical errors will be reported via alerts to ensure swift resolution, while non-critical errors are logged for review.

10. Performance and Scalability

Performance Goals:

Page load time is less than 3 seconds.

Real-time updates achieved by refreshing data every 30 minutes.

Scalability:

Designed to accommodate future growth of news categories through microservices.

DynamoDB supports automatic scaling as data grows.

Disaster Recovery and Backup:
Regular backups of data stored in DynamoDB.
Disaster recovery plan to restore operations within 1 hour in case of failure.

Caching and Optimization:
Uses a Redis cache for frequently accessed articles and API responses to optimize load times.

11. Deployment and Environment

- **Deployment Architecture**:
  The application is designed as a serverless architecture on AWS, enabling scalability and cost-efficiency. The deployment process involves a cloud environment that utilizes several AWS services:
  - **Client Interface**: A front-end built with Vue allows users to interact with the system, sending requests through AWS API Gateway.
  - **API Gateway and Lambda**: API Gateway routes incoming requests to AWS Lambda functions, which handle back-end operations and execute business logic in response to client interactions.
  - **Data Storage and Processing**: AWS DynamoDB serves as the primary NoSQL database for persistent data storage. AWS Batch processes data in bulk when needed, using container images stored in AWS Elastic Container Registry (ECR) to facilitate batch processing and scalability.
  - **Monitoring and Logging**: AWS CloudWatch monitors system health, logs events, and provides performance insights for proactive management.
- **Environment Requirements**:
  - **Development**: Local development environments with AWS SAM or similar tools to simulate AWS Lambda functions and API Gateway locally, ensuring function compatibility.
  - **Testing**: A dedicated testing environment replicates the production setup, allowing for comprehensive testing of Lambda functions, API Gateway routes, and database interactions.
  - **Production**: In production, the application runs fully on AWS, ensuring high availability and reliability with automated scaling managed by AWS services.
- **CI/CD Pipeline**:
  The application follows a continuous integration and delivery (CI/CD) pipeline to streamline deployment and updates. The pipeline includes:
  - **Code Management and Testing**: Code commits trigger automated testing and static code analysis using tools like AWS CodeBuild or GitHub Actions.

- **Deployment**: Upon passing all tests, code changes are deployed to AWS environments using AWS CodePipeline by uploading a new image to Amazon ECR, enabling seamless updates and rollbacks as needed.

This architecture and deployment strategy ensures a robust, scalable, and efficient serverless application that leverages AWS services for real-time processing, secure data handling, and high availability.