Design

Application architecture

Our application is structured around the model-view-controller (MVC) design pattern, which emphasizes modularity and separation of concerns in the realm of web programming.

Conceptually, our application is structured as in Figure 1.

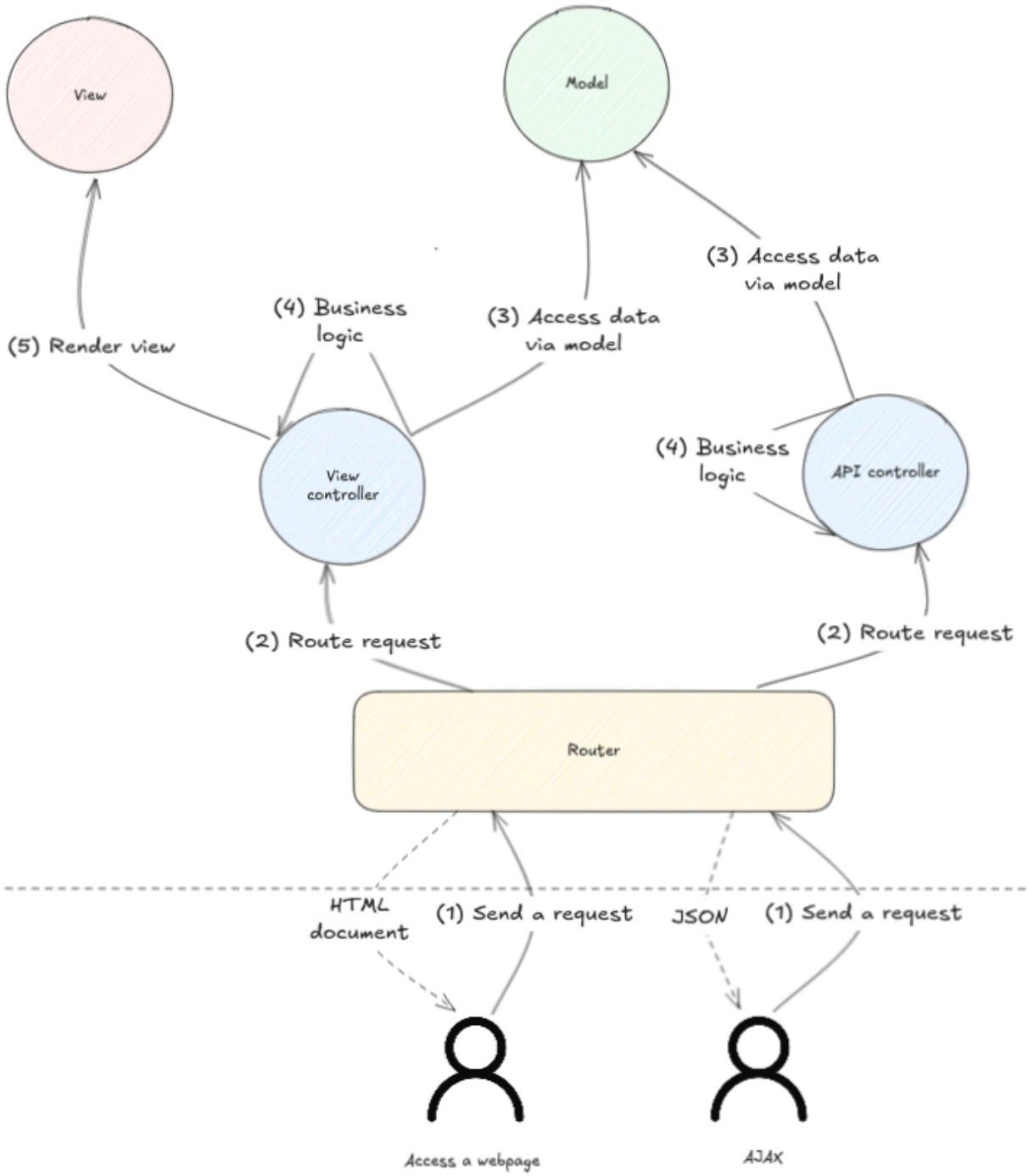

Figure 1: Application structure

There are 4 main components in this figure:

- View: The View component is responsible for presenting data to users through the user interface. It handles the visual aspects of the application, rendering content in HTML format for web

browsers. The View receives processed data from controllers and generates the output that users see and interact with.
- Model: The Model manages the application's data and business logic. It handles data storage, retrieval, and processing operations. It provides a clean interface for accessing application data and enforces data validation rules. The Model contains the core functionality that powers the application's features.
- Controller: Controllers coordinate the interaction between the Model and View components. They process user inputs, manipulate data through the Model, and select which View to present as output. Controllers contain the application's business logic and serve as the central orchestrator in the request-response cycle. The diagram shows specialized controllers for handling both View-based interactions and API requests.
- Router: The Router directs incoming requests to the appropriate controllers based on URL patterns and HTTP methods. It serves as the entry point for all requests to the application, acting as a traffic director that ensures each request reaches the correct handler. The Router enables clean URL structures and separation between different parts of the application.

Source code structure

The physical organization of the codebase reflects the conceptual divisions of MVC presented above, with additional supporting directories for configuration, middleware, and deployment resources:

```
├── api/                 # API endpoints for AJAX interactions
├── config/              # Configuration files and environment settings
├── controllers/         # Controller classes organized by domain
├── middleware/          # Request preprocessing components
├── migrations/          # Database schema version files
├── models/              # Data and business logic components
├── public/              # Publicly accessible assets
├── views/               # Templates and presentation logic
│   ├── layouts/         # Layout templates for page composition
│   └── [feature dirs]/  # Feature-specific view templates
├── .htaccess            # Web server configuration
├── index.php            # Application entry point
└── routes.php           # URL routing definitions
```

This structure balances conceptual purity with practical organization, grouping files by both responsibility and feature context.

In more details, the main files are:
- `api/`: Contains API controller that handle AJAX requests and return structured data in JSON format. These files define interfaces for client-side JavaScript to communicate with the server asynchronously.
- `controllers/`: Houses controller classes that implement the application's core business logic. Controllers are organized by domain or feature and follow a consistent naming convention (e.g., UserController, ProductController). Each controller contains action methods that correspond to specific routes and HTTP methods. Controllers orchestrate the request-response flow by validating input, interacting with models to retrieve or modify data, and selecting appropriate views for rendering responses.
- `routes.php`: Maps all URL patterns and request methods supported by the application and maps them to specific controller actions.
- `models/`: Contains classes that represent business entities and encapsulate data access logic. Models implement validation rules, relationships between entities, and business-specific behaviors.

- `views/`: Stores template files that define the HTML structure and presentation logic. We use our custom templating engine due to the restrictions of the assignment.

Additionally, we have:
- `config/`: Holds environment-specific configuration files for database connections.
- `public/`: : Contains all browser-accessible files including compiled CSS, JavaScript bundles, images, fonts, and other static assets.
- `migrations/`: Stores database migration scripts that define schema changes in a version-controlled, repeatable manner. Each migration file represents a specific change to the database schema.
- `middleware/`: Contains logic that runs before a specific controller for some specific routes.

Application Flow

We define the flow of our application in more details in this section, which is already demonstrated in Figure 1.

.htaccess configuration

For simplicity, we define the followings in our `.htaccess` file:

```
Options -Indexes
FallbackResource /index.php
```

The important directives is `FallbackResource /index.php`. This directive implements URL rewriting for a front controller pattern. It tells the web server that any request that doesn't match an actual file or directory should be handled by `/index.php`. This is crucial for modern web applications with clean URLs because it:
- Creates a single entry point for all application requests.
- Allows the application's router (defined in routes.php) to handle URL interpretation.

Standard Web Request Flow

This flow represents the traditional request/response cycle where the server generates complete HTML pages:

1. **Request Initiation**:
   - User requests a URL through their browser
   - Web server receives the request and routes it to the application entry point
   - `.htaccess` settings ensure all requests are directed to `index.php`.
2. **Request Preprocessing**:
   - Session is initialized.
   - Configuration files are loaded.
   - Routes are registered.
   - View utilities are made available.
   - Middleware is prepared.
3. **Route Resolution**:
   - Application extracts the request path and method.
   - System checks if the path exists in the registered routes.
   - For undefined routes, the system renders a 404 page.
4. **Middleware Processing**:
   - UserMiddleware validates session state and authentication.
   - Middleware may redirect unauthenticated users to login page.
   - Request context is established (e.g., current user).

5. **Controller Delegation**:
   - Appropriate controller is instantiated based on the route.
   - Controller's `route()` method receives HTTP method and path.
   - Controller determines which action method to execute.
6. **Model Interaction**:
   - Controller instantiates required model objects.
   - Models connect to the database through the Database singleton.
   - Models execute queries and retrieve data.
   - Domain objects are created from database records.
   - Business rules are applied to the retrieved data.
7. **View Preparation**:
   - Controller organizes model data into view-friendly structures.
   - Data array is passed to the view rendering function.
8. **View Rendering**:
   - Output buffering is initiated to capture rendered content.
   - View template is included and executes, generating HTML.
   - View may access passed data to populate dynamic elements.
   - Layout template is loaded with the buffered content.
   - Final HTML is sent to the browser.
9. **Response Completion**:
   - Buffer is flushed to the client.
   - Connection is closed.
   - Browser renders the received HTML.

AJAX Interaction Flow

This pattern supports dynamic interactions without requiring full page reloads:

1. **Client-side Initiation**:
   - JavaScript event triggers on user interaction
   - AJAX request is constructed with appropriate headers
   - Request is directed to an API endpoint with payload data
2. **Server Preprocessing**:
   - Request arrives at the application entry point.
   - System detects API path prefix (`/api/`).
   - Direct file inclusion occurs instead of controller routing.
3. **API Endpoint Processing**:
   - Specific API PHP script is loaded.
   - Authentication and authorization are verified.
   - Request parameters are validated.
4. **Model Interaction**:
   - API script interacts with appropriate models.
   - Database operations are performed.
   - Results are processed according to business rules.
5. **Response Formatting**:
   - Data is structured as JSON or required format.
   - Headers are set for content type and caching.

- Response is encoded and sent to client.

6. **Client-side Processing**:
   - JavaScript receives and parses the response.
   - DOM is updated based on received data.
   - User interface reflects the changes without page reload.

This dual-flow architecture provides flexibility to handle both traditional page requests and dynamic interactions while maintaining consistent structural organization and separation of concerns.

Database design

Figure 2 shows the conceptual design of our database via ER diagram.

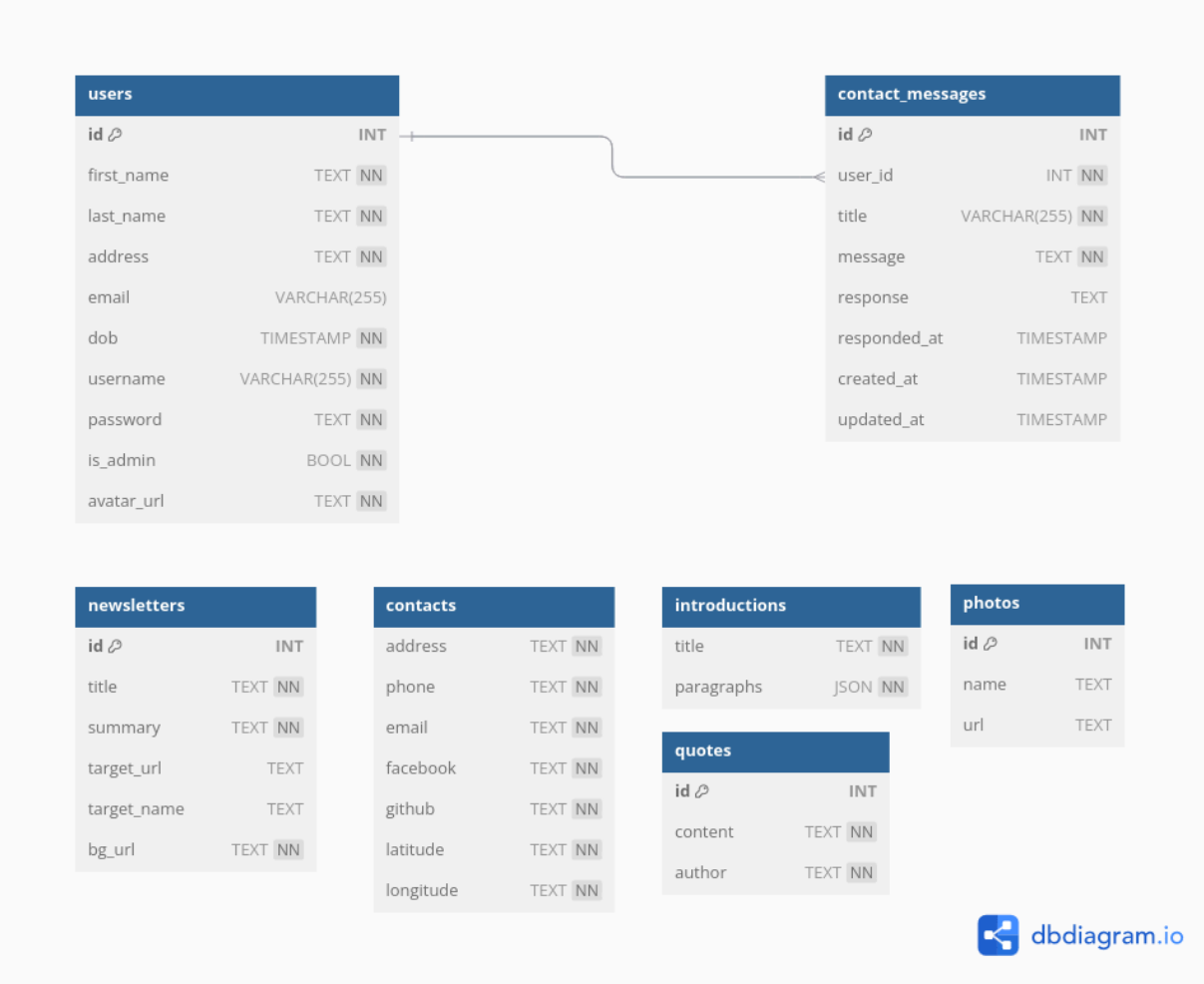

Figure 2: Database design of our application

There are five isolated relations, they are used to store information that's mostly isolated with the rest of the database:

- `quotes`: Remarkable quotes from users to be shown on the homepage.
- `newsletters`: Homepage's sliding banner information.
- `introductions`: A singleton relation with only one row storing the homepage's introduction paragraphs.
- `contacts`: A singleton relation with only one row storing our website's contact info.

There are three main relations. These form the data backbone for our application:

- `users`: A relation storing all users' information.

- `contact_messages`: A relation storing all contact's feedback and reply messages.
- `photos`: A relation storing the photo gallery in our application.