1. What is Vue?

Vue.js is a progressive, open-source JavaScript framework for building user interfaces and single-page applications. It's designed from the ground up to be incrementally adoptable, meaning you can use it to enhance static HTML, embed it as a widget, or build full-scale SPAs. It focuses on the view layer and features a component-based architecture, reactive data binding, and a virtual DOM.

2. Why did you use Vue over React/Jinja2/HTML?

- **Vue vs React:** Often chosen for its perceived simpler learning curve, clearer separation of concerns (HTML templates, JS logic, CSS styling in Single File Components), and sometimes smaller bundle size for less complex applications. While both are powerful, Vue's approach might feel more intuitive to developers familiar with HTML/CSS.
- **Vue vs Jinja2/HTML:** Jinja2 is a server-side templating engine. Using Vue provides client-side rendering capabilities, enabling the creation of dynamic Single Page Applications (SPAs) with faster transitions and richer user interactions without full page reloads. Plain HTML is static; Vue adds reactivity, component reusability, and structured application development needed for complex web interfaces.

3. What are the benefits of Vue over React?

Some perceived benefits (often debated and context-dependent) include:

- **Easier Learning Curve:** Often considered easier for beginners, especially those with existing HTML/CSS knowledge, due to its template syntax.
- **Clearer Separation of Concerns:** Single File Components (`.vue` files) naturally separate template, script, and style.
- **Performance:** While both are highly performant, Vue's reactivity system can sometimes offer better performance out-of-the-box for simple updates compared to React's reconciliation process, although this varies greatly by application complexity.
- **Smaller Bundle Size:** Vue typically has a smaller core library size than React + ReactDOM.

4. Did you use Vue CDN or CLI? Why CLI over CDN?

Usually, for significant projects like a MAD assignment, the **Vue CLI (Command Line Interface)** is used.

- **CDN (Content Delivery Network):** Good for quickly dropping Vue into a simple HTML file or adding interactive features to an existing static site. It loads Vue directly in the browser.

- **CLI:** Used for setting up a full-fledged Vue project. It provides a robust build system (using Webpack or Vite), supports Single File Components (`.vue` files), hot-reloading, linting, testing, and asset management.

Reasons for choosing CLI over CDN:

- **Structure and Organization:** Provides a standard project structure.
- **Single File Components (SFCs):** Allows bundling HTML, CSS, and JavaScript for a component in one file, improving maintainability.
- **Modern JavaScript Features:** Enables using ES6+ features with automatic transpilation.
- **Build Process:** Optimizes code for production (minification, tree-shaking, etc.).
- **Tooling:** Integrates with linters, formatters, testing frameworks.
- **Scalability:** Essential for building larger, maintainable applications.

5. What are Vue lifecycle hooks? Which ones did you use?

Vue components go through a series of stages from creation to destruction. Lifecycle hooks are functions that allow you to execute code at specific points in this process.

Common hooks include:

- `beforeCreate`: Component instance created, but data and events are not yet available.
- `created`: Instance created, data and events available, but DOM not yet mounted. Good for fetching initial data.
- `beforeMount`: Template is compiled/render function created, but not yet mounted to the DOM.
- `mounted`: Component mounted to the DOM. Good for DOM manipulations or integrating third-party libraries that need the DOM.
- `beforeUpdate`: Data changes, before the DOM is re-rendered.
- `updated`: Data changes, after the DOM is re-rendered.
- `beforeUnmount` (Vue 3) / `beforeDestroy` (Vue 2): Component is about to be unmounted/destroyed. Good for cleanup (removing event listeners, timers).
- `unmounted` (Vue 3) / `destroyed` (Vue 2): Component unmounted/destroyed.

Examples of where you might use hooks:

- `created` or `mounted`: To fetch data from a backend API when the component is ready.

JavaScript

```
export default {
  data() {
    return {
      items: []
    };
```

```
  },
  async created() { // Or mounted()
    try {
      const response = await fetch('/api/items');
      this.items = await response.json();
    } catch (error) {
      console.error('Error fetching items:', error);
    }
  }
}
```

- `beforeUnmount`: To clear intervals or remove global event listeners to prevent memory leaks.

JavaScript

```
export default {
  mounted() {
    this.timer = setInterval(() => {
      console.log('Timer running');
    }, 1000);
  },
  beforeUnmount() {
    clearInterval(this.timer);
  }
}
```

(Which specific hooks you used would depend on your project's needs, e.g., fetching data on `created` or `mounted`, cleaning up resources on `beforeUnmount`).

6. Explain how `watch` works in Vue and where you used it.

The `watch` option in Vue allows you to observe changes in a specific data property (or a computed property) and run a callback function when the value changes. It's useful for performing side effects in response to data changes, especially asynchronous or expensive operations.

How it works: You define a `watch` property in your component options, where keys are the names of the data properties (or computed properties) you want to watch, and values are the callback functions or method names to execute.

Example Usage: Watching a search query input to fetch results whenever it changes.

JavaScript

```
export default {
  data() {
    return {
```

```
      searchQuery: '',
      searchResults: []
    };
  },
  watch: {
    searchQuery: {
      // This callback runs whenever searchQuery changes
      async handler(newValue, oldValue) {
        if (newValue.length > 2) { // Fetch only if query is long enough
          console.log(`Search query changed from "${oldValue}" to
"${newValue}"`);
          // Perform API call based on newValue
          try {
            const response = await fetch(`/api/search?q=${newValue}`);
            this.searchResults = await response.json();
          } catch (error) {
            console.error('Search error:', error);
          }
        } else {
          this.searchResults = []; // Clear results if query is too short
        }
      },
      // immediate: true, // Optional: run the handler immediately on component
creation
      // deep: true // Optional: watch nested properties within an object/array
    }
  }
}
```

(Where you used `watch` in your project would be specific to your implementation, e.g., watching form input changes, route parameter changes, or complex object mutations).

7. What is `v-cloak`?

`v-cloak` is a directive used to hide uncompiled Vue.js templates before the Vue instance has finished compiling them. When Vue loads and compiles the template, the `v-cloak` attribute is removed. You typically combine it with CSS rules to hide elements that have this attribute until it's removed.

Example:

HTML

```
<div id="app" v-cloak>
  {{ message }}
</div>
```

CSS

```
[v-cloak] {
  display: none;
}
```

This prevents the user from seeing raw mustache tags (`{{ message }}`) or other uncompiled template syntax briefly during page load.

8. Difference between `v-if` and `v-show`.

Both `v-if` and `v-show` are used for conditionally rendering elements, but they work differently.

- **`v-if` (Conditional Rendering):** `v-if` conditionally **renders** the element and its contents. If the condition is false, the element is not added to the DOM at all. When the condition becomes true, Vue compiles and inserts the element. This is more expensive initially but better for elements that rarely toggle, as it saves on initial rendering cost.
- **`v-show` (Conditional Display):** `v-show` conditionally **displays** the element using CSS. The element is always rendered and kept in the DOM, but its `display` CSS property is toggled between its initial value and none based on the condition. This has higher initial rendering cost but is cheaper to toggle. Better for elements that toggle frequently.

Analogy: `v-if` is like physically adding/removing a piece of furniture from a room. `v-show` is like just hiding/unhiding it behind a curtain.

9. What are slots in Vue? Did you use them?

Slots are a mechanism for content distribution, allowing you to compose components and pass template fragments into a child component from its parent. This makes components more flexible and reusable.

Example: A `Button` component that can have different content (text, icon, etc.) inside it.

Code snippet

```
<template>
  <button class="my-button">
    <slot></slot> </button>
</template>
```

Code snippet

```
<template>
  <div>
    <Button>Click Me</Button> <Button><img src="/icon.png"> Save</Button>
</div>
</template>
```

Slots can also be named (`<slot name="header">`) or scoped (passing data back from the child to the parent's slot content).

(Whether you used them depends on your project's component design, e.g., building layout components, modals, or flexible UI elements).

10. What are props in Vue?

Props (short for properties) are a way of passing data from a parent component down to a child component. They are the primary means of component communication from parent to child.

How they work: A child component declares the props it expects to receive, specifying their name, type, and optionally validation rules or default values. The parent component then binds values to these props when using the child component in its template.

Example: Passing a `title` string and an `isActive` boolean to a `MyComponent`.

Code snippet

```
<template>
  <div>
    <h3>{{ title }}</h3>
    <p v-if="isActive">Status: Active</p>
    <p v-else>Status: Inactive</p>
  </div>
</template>

<script>
export default {
  props: {
    title: {
      type: String,
      required: true
    },
    isActive: {
      type: Boolean,
      default: false
    }
```

```
  }
}
</script>
```

Code snippet

```
<template>
  <div>
    <ChildComponent title="First Item" :isActive="true" />
    <ChildComponent title="Second Item" /> </div>
</template>

<script>
import ChildComponent from './ChildComponent.vue';

export default {
  components: {
    ChildComponent
  }
}
</script>
```

11. How does Vue Router work?

Vue Router is the official routing library for Vue.js. It integrates deeply with Vue to enable building Single Page Applications with navigation powered by URL changes without page reloads.

Key concepts:

- **Routes:** Define mappings between URL paths and Vue components.
- **Router View (`<router-view>`):** A component that renders the matched component for the current route path. It's the placeholder where your route components are displayed.
- **Router Link (`<router-link>`):** A component used for navigation. It renders as an `<a>` tag by default but prevents the browser's default hard reload, instead letting Vue Router handle the navigation internally.
- **History Modes:** Vue Router supports different history modes (e.g., `createWebHistory` for HTML5 History API, `createWebHashHistory` for hash mode) to manage how URLs look and behave.
- **Navigation Guards:** Allow programmatic redirection, cancellation, or modification of navigation requests (e.g., checking authentication before accessing a route).

Basic Flow:

1. User clicks a `<router-link>`.
2. Vue Router intercepts the click, prevents the default browser navigation.
3. Vue Router matches the new URL path to a defined route configuration.
4. Vue Router renders the corresponding component into the `<router-view>` in your main application layout.
5. The URL in the browser changes (in history mode).

12. Why didn't you use Vuex?

(This answer is speculative as I don't know your specific project). Vuex is Vue's official state management library, useful for managing shared application state in large and complex applications. Reasons for *not* using it might include:

- **Project Complexity:** The application was simple enough that state management could be handled locally within components, via props/events, or a simple event bus.
- **Learning Curve/Time Constraints:** Vuex adds a layer of complexity (mutations, actions, getters, modules) that might have been deemed unnecessary for the project scope or limited development time.
- **Alternative Patterns:** For smaller to medium applications, patterns like the Composition API's `provide/inject` (in Vue 3) or a simple global event bus might suffice for sharing state without the full Vuex setup.

13. How to implement template inheritance (like Navbar on every page) in Vue?

In a component-based framework like Vue, you don't typically use traditional template inheritance like Jinja2. Instead, you achieve this through **component composition**.

The most common way is to have a main application layout component (often `App.vue`) or specific layout components that include shared elements like the Navbar, Sidebar, and Footer, and then use `<router-view>` to render the content specific to the current page/route.

Example:

Code snippet

```
<template>
  <div id="app-layout">
    <Navbar /> <main class="content">
      <router-view /> </main>

    <Footer /> </div>
</template>

<script>
```

```
import Navbar from './components/Navbar.vue';
import Footer from './components/Footer.vue';

export default {
  components: {
    Navbar,
    Footer
  }
}
</script>
```

Every route configured in Vue Router will render its component within the `<router-view>`, effectively having the Navbar (and Footer) persist across "pages".

14. What is a for directive in Vue?

The `v-for` directive is used to render a list of items based on an array or an object. It iterates over the source data and creates an element or component for each item.

Syntax: `v-for="item in items"` or `v-for="(item, index) in items"` or `v-for="(value, key, index) in object"`.

Example (Array):

HTML
```
<ul>
  <li v-for="item in items" :key="item.id">
    {{ item.name }}
  </li>
</ul>
```

Example (Object):

HTML
```
<div v-for="(value, key) in myObject" :key="key">
  {{ key }}: {{ value }}
</div>
```

The `:key` attribute is important when using `v-for` with components or maintaining state, helping Vue efficiently update the list.

15. What is a filter in Vue?

In Vue 2, filters were a feature used to apply common text formatting to template expressions. They are typically used at the end of a mustache tag or a `v-bind` expression, denoted by the pipe symbol (|).

Example (Vue 2 syntax):

HTML
```
<p>{{ message | capitalize }}</p>
```

Filters are deprecated in Vue 3. The recommended approach in Vue 3 is to use computed properties or methods for any text formatting logic.

16. Is your project responsive? How did you make it responsive?

(Assuming the project was responsive) Yes, the project was designed to be responsive, meaning the layout and design adapt to different screen sizes (desktops, tablets, mobile phones).

This was achieved using a combination of:

- **CSS Media Queries:** Applying different styles based on screen characteristics like width (`@media (max-width: 768px) { ... }`).
- **Flexible Layouts:** Using CSS Flexbox and Grid for building flexible container layouts that adjust dynamically.
- **Relative Units:** Using relative units like percentages (%), viewport units (vw, vh), and em/rem instead of fixed pixel values for sizes and spacing where appropriate.
- **CSS Framework (Optional):** If a framework like Bootstrap or Tailwind CSS was used, their built-in responsive utility classes and grid systems would be leveraged.

17. What styling did you use (CSS/Bootstrap)?

(State what you actually used) Possible answers:

- **Plain CSS:** Writing custom CSS rules from scratch, potentially organized using methodologies like BEM or SMACSS. With Vue CLI, this often involves writing styles within the `<style>` block of Single File Components, potentially using `scoped` attribute for component-specific styles or preprocessors like SASS/LESS.
- **CSS Framework (Bootstrap/Tailwind/etc.):** Using a pre-built framework like Bootstrap for its grid system, components (navbars, buttons, forms), and utility classes. This speeds up styling but can lead to a more generic look unless customized. Tailwind CSS is another popular utility-first option.

- **Combination:** Using a framework for layout and basic components, then writing custom CSS for specific styling needs or overrides.

18. Explain a component and its fetch request with lifecycle hooks.

A Vue component is a reusable, self-contained block of UI code (template, script, style). It manages its own data, properties (props), and lifecycle.

Consider a `VenueDetails` component that needs to fetch information about a specific venue when it's displayed.

Code snippet

```
<template>
  <div>
    <div v-if="loading">Loading venue details...</div>
    <div v-else-if="error">Error loading venue details:
{{ error.message }}</div>
    <div v-else>
      <h2>{{ venue.name }}</h2>
      <p>{{ venue.location }}</p>
      </div>
  </div>
</template>

<script>
import axios from 'axios'; // Or use native fetch

export default {
  props: {
    venueId: { // Assume venueId is passed from the parent or route params
      type: [String, Number],
      required: true
    }
  },
  data() {
    return {
      venue: null,
      loading: true,
      error: null
    };
  },
  // Lifecycle hook where data fetching typically occurs
  async created() { // 'created' is often preferred if you don't need DOM
```

```
access
    console.log('Component created, fetching data...');
    try {
      const response = await axios.get(`/api/venues/${this.venueId}`);
      this.venue = response.data;
    } catch (err) {
      this.error = err;
      console.error('Error fetching venue:', err);
    } finally {
      this.loading = false;
    }
  },
  // Alternatively, fetch in 'mounted' if you need to access the component's
DOM
  /*
  async mounted() {
     console.log('Component mounted to DOM, fetching data...');
     // Fetch logic similar to 'created'
  },
  */
  beforeUnmount() {
    // Cleanup if necessary, though not typically needed for simple fetch
requests
    console.log('Component before unmount');
  }
}
</script>

<style scoped>
/* Component-specific styles */
</style>
```

In this example:

1. The component receives `venueId` as a prop.
2. In the `data()` function, initial state (`venue: null`, `loading: true`, `error: null`) is defined.
3. The `created` lifecycle hook is used. This hook is called after the instance is created and data observation has been set up, but before the `$el` is mounted to the DOM. It's a suitable place to fetch data because we have access to `this.venueId` and `this.venue`/`this.loading`/`this.error`, but we don't need the DOM yet.

4. Inside `created`, an asynchronous function fetches data from `/api/venues/${this.venueId}`.
5. The `loading` state is updated before and after the fetch.
6. If successful, `this.venue` is updated with the received data.
7. If an error occurs, `this.error` is set.
8. The template conditionally displays loading, error, or venue data based on the state.

Backend & Flask

19. What is Flask SQLAlchemy?

Flask-SQLAlchemy is a Flask extension that simplifies the integration of SQLAlchemy (a powerful Python SQL toolkit and Object-Relational Mapper) with Flask applications. It provides helpers and conventions to make using SQLAlchemy within Flask easier, managing the engine, session, and common patterns like declarative model definition.

20. How to change the database from SQLite to another (required code changes)?

Changing from SQLite to another database like PostgreSQL or MySQL typically involves:

1. **Install Database Adapter:** Install the appropriate Python library for the new database (e.g., `psycopg2` for PostgreSQL, `mysql-connector-python` or `PyMySQL` for MySQL).
2. **Update Configuration:** Change the `SQLALCHEMY_DATABASE_URI` in your Flask configuration to the connection string for the new database.
    a. SQLite: `sqlite:///path/to/your/database.db`
    b. PostgreSQL: `postgresql://user:password@host:port/database`
    c. MySQL: `mysql+mysqlconnector://user:password@host:port/database`
3. **Database Setup:** Ensure the new database server is running and the database/user exist with appropriate permissions.
4. **Run Migrations:** If you used a migration tool like Flask-Migrate (which uses Alembic), you would generate a new migration script that reflects the models and then apply it to the new empty database. This creates the necessary tables and columns.

Bash
```
flask db migrate -m "Migrate to new database"
flask db upgrade
```

5. **Check for Database-Specific Syntax:** While SQLAlchemy aims for database agnosticism, complex queries or specific features might occasionally require minor adjustments based on the target database's SQL dialect.

21. What is ORM?

ORM stands for Object-Relational Mapper. It's a technique that allows you to interact with a relational database using an object-oriented programming language (like Python) instead of writing raw SQL. An ORM maps database tables to classes (models), rows to objects, and columns to object attributes.

Benefits:

- **Abstraction:** Hides database-specific SQL syntax.
- **Increased Productivity:** Allows developers to work with familiar objects.
- **Maintainability:** Code is often more readable and easier to manage.
- **Portability:** Can potentially switch databases with minimal code changes (if using a database-agnostic ORM).

SQLAlchemy is a popular ORM in the Python world.

22. How do you handle authentication in Flask?

Authentication (verifying a user's identity) in Flask is commonly handled using:

1. **Manual Sessions:** Storing user ID in the Flask session (`session['user_id'] = user.id`) after successful login. Middleware or decorators check for the presence of this session key on protected routes.
2. **Flask-Login Extension:** This is the standard way. It provides helpers for managing user sessions, handling logins/logouts, remembering users, and restricting access to routes.
    a. Define a User model that implements Flask-Login's `UserMixin`.
    b. Implement a `user_loader` function to load a user from the ID stored in the session.
    c. Use the `login_user()` function after successful login.
    d. Use the `logout_user()` function.
    e. Use the `@login_required` decorator to protect routes.
    f. Implement password hashing (e.g., using `werkzeug.security.generate_password_hash` and `check_password_hash`) for secure password storage.

23. What is `login_required` decorator? Explain its implementation.

`@login_required` is a decorator provided by the Flask-Login extension. Its purpose is to protect routes (view functions) that should only be accessible to authenticated users.

Implementation: You apply the decorator directly above the view function definition.

Python

```
from flask import Flask, render_template, redirect, url_for
from flask_login import LoginManager, login_user, logout_user, login_required,
UserMixin, current_user
from werkzeug.security import generate_password_hash, check_password_hash

# ... (Flask app setup)
# ... (User model definition inheriting UserMixin)
# ... (login_manager setup and user_loader)

@app.route('/dashboard')
@login_required # This decorator is applied here
def dashboard():
    # This code will only execute if the user is logged in
    return render_template('dashboard.html', user=current_user)

@app.route('/profile')
@login_required # This decorator is applied here
def profile():
    # This code will only execute if the user is logged in
    return render_template('profile.html', user=current_user)

@app.route('/login', methods=['GET', 'POST'])
def login():
    # ... (Handle login form submission)
    if login_successful:
        user = User.query.filter_by(email=email).first()
        login_user(user) # Log the user in
        return redirect(url_for('dashboard'))
    # ... (Render login form)

@app.route('/logout')
@login_required # You can require login to logout, or allow anyone to try
def logout():
    logout_user() # Log the user out
    return redirect(url_for('index')) # Redirect to home page
```

When an unauthenticated user tries to access a route decorated with `@login_required`, Flask-Login intercepts the request and typically redirects them to the login view (configured via `login_manager.login_view`). The originally requested URL is often stored in a query parameter (`next`) so the user can be redirected back after logging in.

24. How is RBAC (Role-Based Access Control) implemented?

RBAC involves assigning permissions to roles, and then assigning roles to users. Users inherit the permissions of their assigned roles.

Implementation approaches in Flask:

1. **Manual Checks:**
    a. Add a `role` column (e.g., string or foreign key to a `Role` table) to the User model.
    b. In view functions or decorators, check the user's role (`if current_user.role == 'admin':`). This can become cumbersome for complex permission logic.
2. **Using Extensions (e.g., Flask-Principal, Flask-Security-Too):**
    a. **Flask-Principal:** Defines "identities" (users) and "permissions". Permissions are granted to identities based on "principals" (like roles). You define `Permission` objects and use them to check access.
    b. **Flask-Security-Too:** Provides a more complete authentication and authorization solution, including built-in support for roles and role-checking decorators (`@roles_required`, `@roles_accepted`).
    c. Typically involves `User`, `Role`, and a many-to-many association table between them if users can have multiple roles. Permissions are associated with roles.

Example (Manual with separate Role table):

Python
```
# models.py
class Role(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(80), unique=True)
    users = db.relationship('User', backref='role', lazy='dynamic') # Assuming
one role per user

class User(db.Model, UserMixin):
    id = db.Column(db.Integer, primary_key=True)
    # ... other user fields
    role_id = db.Column(db.Integer, db.ForeignKey('role.id'))

# views.py
from flask import abort
from flask_login import login_required, current_user

@app.route('/admin/dashboard')
@login_required
def admin_dashboard():
```

```
if current_user.role.name != 'admin':
    abort(403) # Forbidden
# Admin logic here
return render_template('admin_dashboard.html')
```

25. Difference between authentication and authorization.

- **Authentication:** The process of verifying the identity of a user or system. It answers the question "Who are you?". Examples: logging in with a username and password, using a digital certificate.
- **Authorization:** The process of determining what an authenticated user or system is allowed to do. It answers the question "What are you allowed to do?". Examples: An admin user is authorized to delete users, a regular user is not. RBAC is a method of implementing authorization.

26. What is CORS?

CORS stands for Cross-Origin Resource Sharing. It is a security mechanism implemented by web browsers that prevents a web page from making requests to a domain that is different from the one the page originated from (the "Same-Origin Policy").

CORS defines a way for the browser and the server to communicate and determine whether the cross-origin request is allowed. The server needs to send specific HTTP headers (like `Access-Control-Allow-Origin`) in its response to indicate which origins are permitted to access its resources.

In a Flask backend serving an API to a separate Vue.js frontend running on a different port or domain, CORS must be configured on the Flask server to allow the frontend domain to make requests. Flask-CORS is an extension that simplifies this.

27. What is CSRF? How is it handled?

CSRF stands for Cross-Site Request Forgery. It is an attack where a malicious website or email tricks a user's browser into making an unwanted request to a web application where the user is currently authenticated. Because the user is logged in, the application treats the malicious request as legitimate.

How it's handled/prevented: The most common defense is using **CSRF tokens**.

1. When a sensitive form or request is sent from the legitimate application, the server includes a unique, unpredictable token (the CSRF token) with the form or request data (e.g., in a hidden input field or a custom HTTP header).

2. When the request is submitted, the server verifies that the received token matches the one it expected (e.g., stored in the user's session).
3. If the tokens don't match, the server rejects the request. Since the malicious site cannot obtain the valid token (due to the Same-Origin Policy for reading data), it cannot successfully forge the request. Flask-WTF and Flask-Security-Too provide easy ways to integrate CSRF protection.

28. How are tokens managed (JWT/OAuth)?

Tokens (like JWTs) are often used for stateless authentication, especially in API-driven applications where sessions might be less suitable.

JWT (Json Web Tokens):

1. **Login:** User authenticates (e.g., username/password).
2. **Server Issues Token:** If credentials are valid, the backend generates a JWT containing claims (user ID, roles, expiration time, etc.) and signs it with a secret key. The server sends this JWT back to the client.
3. **Client Stores Token:** The client (e.g., Vue.js app) receives the JWT and typically stores it in browser storage (like `localStorage` or cookies, though `localStorage` is common for SPAs).
4. **Client Sends Token:** For subsequent requests to protected API endpoints, the client includes the JWT, usually in the `Authorization` header as a `Bearer` token (`Authorization: Bearer <your_jwt>`).
5. **Server Validates Token:** The backend receives the request, extracts the JWT from the header, verifies its signature using the secret key (to ensure it hasn't been tampered with), and checks the claims (e.g., checks if the token has expired, gets the user ID). If valid, it processes the request. The server doesn't need to look up session data, making it stateless.

OAuth (Authorization Framework): OAuth is primarily for *authorization* (granting limited access to user data without sharing credentials), often used for "Login with Google/Facebook". It involves multiple parties (user, client application, authorization server, resource server) and flows (Authorization Code Flow, etc.) to issue access tokens. These access tokens grant permission to the client app to access specific resources on behalf of the user. Management involves redirects, getting authorization codes, exchanging codes for tokens, and using tokens to access APIs.

In summary, tokens are managed by the server issuing them upon successful authentication/authorization, and the client storing and including them in subsequent requests, which the server then validates.

29. Is Flask open-source? Can it be used for large applications?

- Yes, **Flask is open-source**. It is licensed under the BSD license.

- Yes, Flask **can be used for large applications**. While it's a "microframework" (meaning it provides a core, minimal set of features), its modular design and extensive ecosystem of extensions allow it to be scaled and structured for complex projects. For large applications, careful architecture is needed, often involving Blueprints (for modularity), extensions for databases, authentication, etc., and potentially separating concerns into microservices.

30. What is lazy loading?

Lazy loading is a technique used to defer the loading or initialization of a resource or object until it is actually needed. This can improve performance by reducing initial load times and resource consumption.

Examples:

- **Database ORMs (like SQLAlchemy):** Related objects (e.g., the `users` relationship on a `Role` object) are often lazy-loaded by default. When you access `role.users`, SQLAlchemy performs a separate query to fetch the associated users *only at that moment*, rather than loading all related users when the role object itself is fetched.
- **Frontend (Vue.js):** Lazy loading routes or components means the JavaScript code for a specific page or component is only downloaded and parsed by the browser when the user navigates to that page or when the component is needed. This reduces the initial bundle size and speeds up the initial page load. Vue Router supports lazy loading components using dynamic imports (`component: () => import('./views/About.vue')`).

31. How to index in SQLAlchemy?

Database indexing is used to improve the speed of data retrieval operations (SELECT queries) at the cost of slower data modification operations (INSERT, UPDATE, DELETE) and increased storage space.

In SQLAlchemy, you can create indexes in a few ways:

1. **Column-Level Index:** Adding `index=True` to a column definition creates a single-column index on that column.

Python

```
class User(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    email = db.Column(db.String(120), unique=True, index=True) # Indexed column
    username = db.Column(db.String(80), unique=True)
```

2. **Table-Level Index (for composite or advanced indexes):** Using the `Index` object, typically within the `__table_args__` of the model class. This is used for creating indexes on multiple columns or specifying specific index types/configurations.

Python

```
from sqlalchemy import Index

class Order(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    user_id = db.Column(db.Integer, db.ForeignKey('user.id'))
    order_date = db.Column(db.DateTime)
    status = db.Column(db.String(50))

    __table_args__ = (
        # Composite index on user_id and order_date for faster lookups
        Index('ix_user_order_date', 'user_id', 'order_date'),
        # Index on status, potentially partial index depending on DB dialect
        Index('ix_status', 'status'),
    )
```

After defining indexes in your models, you would typically use a database migration tool (like Flask-Migrate/Alembic) to apply these schema changes to your database.

Celery & Redis

32. What is Celery? Explain its components.

Celery is an asynchronous task queue/job queue based on distributed message passing. It's used to run tasks in the background outside the main application flow, typically for long-running or resource-intensive operations that would block the main application or require scheduling.

Key Components:

1. **Producer (or Client):** The application code that creates and sends tasks to the task queue. In a Flask app, this would be your web view or service that calls a `.delay()` or `.apply_async()` method on a Celery task.
2. **Broker (or Message Queue):** Acts as an intermediary between the producer and the workers. It stores the tasks in a queue and delivers them to workers. Common brokers include Redis, RabbitMQ, and Amazon SQS. Celery needs a broker to operate.
3. **Worker:** Processes that consume tasks from the broker's queue and execute them. You typically run one or more worker processes. Workers need to have access to the task code.
4. **Backend (or Result Backend):** (Optional) Used to store the results of completed tasks or task states. Can be a database (like SQLAlchemy), Redis, or other storage systems. Useful if you need to check the status or result of a task from your web application.

33. What Celery jobs have you implemented?

(State the specific background tasks implemented in your project). Common examples include:

- **Sending emails:** Sending welcome emails, password reset emails, booking confirmations.
- **Generating reports:** Creating CSV or PDF reports of venue/booking data.
- **Data processing:** Resizing images, processing uploaded files.
- **Background calculations:** Performing complex calculations that don't need immediate user response.
- **Scheduled tasks:** Tasks run at fixed intervals (using Celery Beat).

Example: A task to generate a venue report CSV.

Python

```
# tasks.py
from celery import Celery
# ... setup Celery app

@celery.task
def generate_venue_report_csv(venue_id):
    # ... fetch venue data and related bookings from database
    # ... generate CSV content
    # ... save CSV to a file or cloud storage
    # ... return path to file or a confirmation
    pass

# In your Flask view/service:
@app.route('/venue/<int:venue_id>/report')
@login_required
def generate_report_view(venue_id):
    if not current_user.is_admin:
        abort(403)
    # Enqueue the task
    task = generate_venue_report_csv.delay(venue_id)
    # Return a response indicating the report is being generated
    return jsonify({"message": "Report generation started", "task_id":
task.id}), 202
```

34. Explain Celery Beat and scheduled tasks.

Celery Beat is a scheduler component for Celery. It reads a schedule (defined in Celery's configuration) and sends tasks to the broker at the specified times or intervals. It's essentially like a fault-tolerant cron job system for Celery tasks.

Scheduled Tasks: These are tasks configured in Celery Beat to run periodically or at specific times. You define the task name and the schedule (e.g., run `cleanup_old_data` every night at 2 AM, run `send_daily_summary` every day at 9 AM).

Example Configuration (in Celery config):

Python
```
# celeryconfig.py or app config
CELERY_BEAT_SCHEDULE = {
    'send-report-every-monday': {
        'task': 'tasks.generate_weekly_summary_report',
        'schedule': crontab(hour=9, minute=0, day_of_week=1), # Every Monday at
9:00 AM
        'args': (123,) # Optional arguments for the task
    },
    'cleanup-every-night': {
        'task': 'tasks.cleanup_expired_sessions',
        'schedule': timedelta(hours=24), # Every 24 hours (daily)
    },
}
```

To use Celery Beat, you need to run a separate Celery Beat process in addition to the workers: `celery -A your_celery_app beat`.

35. How to shut down a Celery task without using the terminal?

You can shut down or stop a running Celery task programmatically using Celery's control commands. This is done by obtaining the `AsyncResult` object for the task (using its `task_id`) and calling its `revoke()` method.

Python
```
# Example in a Flask view or background process
from celery.result import AsyncResult
from your_celery_app import celery # Assuming you have your Celery app instance

@app.route('/cancel-task/<task_id>')
@login_required # Protect this endpoint
```

```
def cancel_task(task_id):
    if not current_user.is_admin:
        abort(403)

    task = AsyncResult(task_id, app=celery)
    if task.state not in ['PENDING', 'RECEIVED', 'STARTED']:
         return jsonify({"message": f"Task {task_id} is already in state
{task.state}, cannot revoke."}), 400

    # Revoke the task
    # terminate=True is for forceful termination (SIGKILL), use with caution
    # default behavior (terminate=False, signal='SIGTERM') is a graceful
shutdown request
    task.revoke(terminate=True)
    return jsonify({"message": f"Revoked task {task_id}"}), 200
```

This allows you to build a UI or API endpoint to manage running tasks. Note that graceful termination (`SIGTERM`) relies on the task code handling the signal. Forceful termination (`SIGKILL`) immediately kills the worker process handling the task, which might lead to data corruption if not handled carefully.

36. What is Redis? Why use it over a normal database?

Redis (Remote Dictionary Server) is an open-source, in-memory data structure store used as a database, cache, and message broker.

Why use it over a normal (disk-based) database like PostgreSQL or MySQL?

- **Speed:** Being primarily in-memory, Redis offers significantly faster read and write operations compared to disk-based databases, making it ideal for caching and real-time data.
- **Data Structures:** Redis supports various advanced data structures like lists, sets, sorted sets, hashes, and geospatial indexes, which go beyond the simple key-value store of some caching systems and are often more efficient for certain use cases than modeling them relationally.
- **Specific Features:** Provides built-in features like publish-subscribe (for messaging), transactions, Lua scripting, and optional persistence (saving data to disk periodically).
- **Caching:** Its speed and in-memory nature make it an excellent choice for a cache layer to reduce the load on the primary database.

Normal databases are designed for durable storage, complex queries with relationships, and transactional integrity on disk, while Redis excels at high-speed access to volatile or semi-volatile data and specific data manipulation patterns.

37. Difference between Memcached and Redis.

Memcached and Redis are both popular in-memory key-value stores often used for caching, but Redis is generally more feature-rich.

- **Data Structures:**
  - Memcached: Primarily a simple key-value store (strings).
  - Redis: Supports a wider variety of data structures (strings, lists, sets, sorted sets, hashes, bitmaps, hyperloglogs).
- **Persistence:**
  - Memcached: No built-in persistence; data is lost if the server restarts.
  - Redis: Supports persistence options (RDB snapshots and AOF log) to save data to disk.
- **Features:**
  - Memcached: Basic get/set/delete operations.
  - Redis: More advanced features like publish-subscribe, transactions, Lua scripting, expirations, atomic operations.
- **Replication & Clustering:** Redis has robust built-in support for replication and clustering. Memcached clustering is typically client-side or via external tools.

Redis is often chosen when more than just simple key-value caching is needed, or when persistence, pub/sub, or richer data structures are beneficial. Memcached is still popular for very simple, high-performance caching where memory efficiency is paramount and the richer features of Redis aren't required.

38. What is caching? Where did you implement it?

Caching is a technique where copies of frequently accessed data or results of expensive computations are stored in a temporary, faster storage location (the cache). When the data is needed again, the application first checks the cache. If the data is found in the cache ("cache hit"), it's retrieved quickly without re-computing or re-fetching from the original, slower source. If not ("cache miss"), the data is fetched from the source, used, and then stored in the cache for future requests.

Purpose: To improve performance, reduce latency, and decrease the load on backend services, databases, or external APIs.

(Where you implemented it would depend on your project). Common places to implement caching in a web application:

- **Database Query Caching:** Caching the results of frequently run database queries (e.g., using Redis to store results of complex joins or aggregate queries).
- **API Response Caching:** Caching the responses from external APIs or even your own backend API endpoints.
- **Template Fragment Caching:** Caching parts of rendered HTML templates.

- **Function/Method Result Caching (Memoization):** Caching the return values of specific functions based on their input parameters (e.g., using `@cached` decorator in Python, similar to memoization).
- **Browser Caching:** Using HTTP headers (`Cache-Control`, `Expires`, `ETag`) to instruct the browser to cache static assets (CSS, JS, images) or API responses.

(Provide specific examples from your project, e.g., "I cached the results of the query fetching all venues using Redis to reduce database load on the homepage" or "I used browser caching for static assets").

39. What is publish-subscribe in Redis?

Publish-Subscribe (Pub/Sub) is a messaging pattern where senders (publishers) do not send messages directly to specific receivers (subscribers). Instead, they publish messages to a channel. Subscribers express interest in one or more channels and receive all messages published to the channels they are subscribed to.

Redis supports Pub/Sub natively.

- **PUBLISH command:** A client publishes a message to a specified channel.
- **SUBSCRIBE command:** A client subscribes to one or more channels.
- **PSUBSCRIBE command:** A client subscribes to channels matching a pattern.

Use Cases:

- **Real-time Updates:** Notifying connected clients (e.g., web browsers via WebSockets) about events, like new messages, data changes, or task completion status.
- **Decoupling Services:** Allowing different parts of your application or different microservices to communicate asynchronously without needing direct knowledge of each other. A service publishes an event (e.g., "order completed"), and other services interested in that event (e.g., inventory service, email service) subscribe to the channel and react accordingly.

Security

40. What is a rainbow table?

A rainbow table is a precomputed table used to reverse cryptographic hash functions, typically used to find the original password from a stored password hash. It works by pre-calculating hash chains (sequences of hashing and reduction functions) for a large number of potential passwords. Attackers can look up a target hash in the table to find a corresponding hash chain that leads back to a potential original password.

Defense: Salting passwords. A unique, random value (the salt) is added to the password *before* hashing. The salt is stored alongside the hash. Because each password now has a unique salted hash, rainbow tables (which are built on precomputed hashes for *unsalted* passwords) are ineffective.

41. What was the default hashing algorithm in MAD1?

(Self-correction: I don't know the specifics of a particular course named MAD1. I will explain common hashing practices for passwords and mention SHA-256, which might have been used, but also discuss its limitations for passwords without proper application.)

If "MAD1" involved basic password hashing, it *might* have used an algorithm like SHA-256 or MD5. However, **MD5 and raw SHA-256 are NOT suitable for password hashing** because they are too fast (making brute-force attacks easier) and vulnerable to rainbow tables (especially if unsalted).

Modern password hashing algorithms are designed to be computationally expensive and include salting inherently. These include:

- **bcrypt**
- **scrypt**
- **Argon2** (currently recommended)
- **PBKDF2**

If SHA-256 *was* used in MAD1 for passwords, it should ideally have been used with multiple iterations (stretching) and salting to increase the computational cost and resist rainbow tables. Using raw, unsalted SHA-256 for passwords would be a security vulnerability.

42. What is SHA-256? Can it be decrypted?

SHA-256 (Secure Hash Algorithm 256-bit) is a cryptographic hash function, part of the SHA-2 family. It takes an input (any data) and produces a fixed-size 256-bit (32-byte) output hash value, often represented as a 64-character hexadecimal string.

Properties of a good hash function like SHA-256:

- **Deterministic:** The same input always produces the same output hash.
- **Fast Computation:** Relatively quick to compute the hash for any given input.
- **Pre-image Resistance (One-way):** It's computationally infeasible to find the original input given only the output hash.
- **Second Pre-image Resistance:** It's computationally infeasible to find a *different* input that produces the same hash as a given input.
- **Collision Resistance:** It's computationally infeasible to find two *different* inputs that produce the same output hash.

Can it be decrypted? NO. Hash functions are one-way functions. They are not designed for encryption (which is two-way: encrypt and decrypt). You cannot mathematically reverse a SHA-256 hash to get the original data back.

Attacks on hashes (like rainbow tables or brute force) don't "decrypt" the hash; they try to find an input that *produces* the target hash by trying many possibilities or using precomputed tables.

43. How many hashing algorithms do you know?

(List the algorithms you are familiar with from various contexts). Examples:

- **Cryptographic Hash Functions (for data integrity, digital signatures):** MD5 (collision vulnerabilities, generally not recommended for security), SHA-1 (weaknesses found, being deprecated), SHA-2 family (SHA-256, SHA-512), SHA-3 family.
- **Password Hashing Functions (designed to be slow and include salting):** bcrypt, scrypt, Argon2 (recommended), PBKDF2.
- **Non-Cryptographic Hashes (for data structures, checksums):** MurmurHash, CityHash, etc.

You should be able to explain the *purpose* and *suitability* of different types (e.g., why password hashes are different from general-purpose hashes).

44. How to hide passwords in network payloads?

Passwords should **never** be sent in plain text over a network, regardless of whether they are "hidden" within a payload. The standard and essential way to protect passwords (and all sensitive data) in transit is by using **HTTPS (HTTP over TLS/SSL)**.

HTTPS encrypts the entire communication channel between the user's browser and the server. Even if an attacker intercepts the network traffic (Man-in-the-Middle attack), they cannot read the actual data (including the password) because it is encrypted.

On the server side, you never store the plain text password. You store a secure hash of the password (using algorithms like bcrypt, scrypt, or Argon2) along with a unique salt. When a user tries to log in, you hash the entered password with the stored salt and compare the resulting hash to the stored hash.

45. What is SQL injection? How to prevent it?

SQL Injection is a web security vulnerability where an attacker is able to interfere with the queries that an application makes to its database. It typically involves injecting malicious SQL code into user input fields that are not properly sanitized or parameterized.

How it works: If an application directly concatenates user input into an SQL query string, an attacker can input data that changes the query's logic.

- *Vulnerable code:* `query = "SELECT * FROM users WHERE username = '" + username + "' AND password = '" + password + "';`
- *Malicious input for username:* `' OR '1'='1`
- *Resulting query:* `SELECT * FROM users WHERE username = '' OR '1'='1' AND password = '...'` - The `' OR '1'='1'` part makes the WHERE clause always true, potentially allowing the attacker to log in without a password or retrieve all user data.

How to prevent it: The primary defense is to use **parameterized queries (or prepared statements)**. This involves separating the SQL command from the data values. The database engine is told to treat the data values strictly as data, not as executable SQL code.

- When using ORMs like SQLAlchemy, they handle this automatically by default when you use their query building methods (e.g., `session.query(User).filter_by(username=username).first()`).
- When writing raw SQL with database connectors (like `psycopg2`), you pass the SQL string and the data values separately to the execute method, letting the library handle the parameterization.

Other defenses include input validation (though not sufficient on its own) and using least privilege for database users.

46. What is a man-in-the-middle attack?

A Man-in-the-Middle (MITM) attack is a type of cyberattack where an attacker intercepts communication between two parties who believe they are communicating directly with each other. The attacker can then eavesdrop on the conversation, steal information, or even alter the communication before relaying it.

Example: An attacker positioning themselves between a user's browser and a website's server. Without proper security measures, the attacker can read sensitive data (like passwords) being exchanged or inject malicious content into the communication.

Prevention: The primary defense against MITM attacks on the web is using **HTTPS/SSL/TLS encryption**. This encrypts the data exchanged between the browser and server, making it unreadable to anyone intercepting the traffic. Ensuring valid SSL certificates are used is crucial to verify the identity of the server.

47. How can you improve the security of your application?

Security is an ongoing process. Improvements can include:

- **Input Validation & Sanitization:** Thoroughly validate and sanitize all user inputs to prevent injections (SQL, XSS) and other vulnerabilities.

- **Secure Password Handling:** Use strong, modern hashing algorithms (Argon2, bcrypt) with unique salts. Enforce password complexity rules. Implement account lockout after failed attempts.
- **Use HTTPS:** Always encrypt communication using SSL/TLS.
- **Implement CSRF Protection:** Use tokens for sensitive actions.
- **Implement XSS Protection:** Sanitize output to prevent Cross-Site Scripting. Set appropriate HTTP headers (`X-Content-Type-Options`, `X-Frame-Options`, `X-XSS-Protection`, `Content-Security-Policy`).
- **Secure Session Management:** Use secure, HTTP-only cookies for session IDs. Regenerate session IDs upon login. Set appropriate expiration times.
- **Implement Authentication & Authorization Properly:** Use strong authentication methods and robust RBAC. Apply the principle of least privilege.
- **Keep Dependencies Updated:** Regularly update frameworks, libraries, and extensions to patch known vulnerabilities.
- **Error Handling:** Avoid revealing sensitive information in error messages (e.g., database details, stack traces).
- **Logging and Monitoring:** Log security-relevant events (failed logins, access attempts) and monitor logs for suspicious activity.
- **Security Audits & Testing:** Regularly review code for vulnerabilities and perform penetration testing.
- **Rate Limiting:** Protect against brute force and denial-of-service attacks on endpoints like login.

48. Difference between security and privacy.

- **Security:** Focuses on protecting data, systems, and networks from unauthorized access, use, disclosure, disruption, modification, or destruction. It's about protecting *against* threats and ensuring confidentiality, integrity, and availability of information.
- **Privacy:** Focuses on an individual's right to control their personal information. It's about how data is collected, stored, used, and shared, and ensuring that individuals have control over their own data.

Security is often a *means* to achieve privacy. For example, encrypting personal data (security measure) helps ensure that only authorized individuals can access it, thus protecting the individual's privacy. However, you can have a secure system that doesn't respect privacy (e.g., collecting excessive personal data securely) or a system that aims for privacy but lacks sufficient security measures to protect the private data.

APIs & HTTP Methods

49. Difference between POST, PUT, and PATCH.

These are common HTTP methods used to send data to a server, typically for creating or updating resources in a RESTful API context.

- **POST:** Used to **create a new resource**. It is non-idempotent, meaning multiple identical POST requests will likely result in multiple resources being created (e.g., submitting a form twice creates two identical entries). The endpoint often represents a collection (e.g., `POST /users` to create a new user).
- **PUT:** Used to **update/replace an existing resource or create a new resource if it doesn't exist**. It is idempotent, meaning multiple identical PUT requests to the *same URL* should have the same effect as a single request (the resource is replaced with the same data each time). The URL typically specifies the resource being acted upon (e.g., `PUT /users/123` to replace the user with ID 123). The request body should contain the complete representation of the resource after the update.
- **PATCH:** Used to **apply partial modifications to an existing resource**. It is non-idempotent. The request body contains instructions describing the changes to be made, rather than the complete new state of the resource (e.g., `PATCH /users/123` with `{ "email": "new@example.com" }` to change only the email).

50. Can POST be used instead of PUT?

Yes, technically **POST is often used instead of PUT** in practice, especially in situations where PUT's idempotency or the requirement to send the full resource representation is inconvenient or not fully understood. POST is a more general-purpose method for sending data to a resource, and servers will often accept POST requests for updates.

However, from a strict RESTful design perspective and for semantic clarity, using PUT to replace a resource and PATCH to partially update a resource is preferred because it better describes the intended action and leverages the specific properties of those methods (like idempotency for PUT). Using POST for all creation and update operations can make an API less clear and harder for clients to interpret the intended side effects.

51. Difference between GET and POST.

These are two fundamental HTTP methods.

- **GET:** Used to **request data from a specified resource**.
    - Parameters are sent in the URL query string (`/users?id=123`).

    - Requests should only retrieve data and have no side effects on the server (should be safe and idempotent).
    - Requests can be cached by browsers and proxies.
    - Requests remain in browser history and logs.
    - There are URL length limitations.
- **POST:** Used to **send data to a server to create or update a resource**.
    - Parameters are sent in the request body.
    - Requests can have side effects on the server (e.g., creating a new database record).
    - Requests are not typically cached.
    - Requests are not bookmarkable.
    - No practical limit on data size sent in the body.

Use GET for fetching data (like viewing a profile) and POST for submitting data that causes a change on the server (like creating a new user or submitting a form).

52. How are API headers/auth tokens sent to the API?

HTTP headers are key-value pairs included in the request or response. Authentication tokens (like JWTs) are typically sent from the client to the server in the **`Authorization`** HTTP header.

Standard format: `Authorization: <Type> <Credentials>`

For most token-based authentication schemes (including JWT), the type is `Bearer`, followed by the token.

Example using a Fetch API call in JavaScript:

JavaScript

```
const token = localStorage.getItem('accessToken'); // Assuming the token is
stored here

fetch('/api/protected-resource', {
  method: 'GET', // Or POST, PUT, etc.
  headers: {
    'Authorization': `Bearer ${token}`, // The Authorization header with the
token
    'Content-Type': 'application/json' // Example of another common header
    // ... other headers
  }
})
.then(response => {
  if (!response.ok) {
    if (response.status === 401 || response.status === 403) {
      console.error('Authentication or authorization failed');
```

```
      // Redirect to login page
    }
    throw new Error(`HTTP error! status: ${response.status}`);
  }
  return response.json();
})
.then(data => {
  console.log(data);
})
.catch(error => {
  console.error('Fetch error:', error);
});
```

The backend server receives this request, extracts the token from the `Authorization` header, validates it, and if valid, processes the request.

Caching & Performance

53. What is memoization?

Memoization is an optimization technique used to speed up computer programs by storing the results of expensive function calls and returning the cached result when the same inputs occur again. It's applicable to "pure" functions – functions that always produce the same output for the same input and have no side effects.

How it works:

1. The function checks if the result for the current set of input arguments is already stored in a cache (often a hash map or dictionary).
2. If the result is found (cache hit), it returns the cached result immediately.
3. If the result is not found (cache miss), the function executes its logic, computes the result, stores the result in the cache associated with the input arguments, and then returns the result.

Example (Conceptual Python):

Python

```
# Without memoization
def fibonacci(n):
  if n <= 1:
    return n
  return fibonacci(n - 1) + fibonacci(n - 2) # Redundant calculations
```

```
# With memoization (simplified)
def fibonacci_memoized(n, cache={}):
  if n in cache:
    return cache[n]
  if n <= 1:
    return n
  result = fibonacci_memoized(n - 1, cache) + fibonacci_memoized(n - 2, cache)
  cache[n] = result # Store result
  return result
```

Libraries often provide decorators for memoization (e.g., `functools.lru_cache` in Python).

54. Difference between memoization and caching.

Memoization is a **specific form of caching** applied to the results of functions based on their input arguments. It's typically implemented at the function or method level in code.

Caching is a **broader concept** that involves storing any kind of data (database query results, API responses, static files, computed values, function results) in a faster access layer to avoid re-fetching or re-computing them from a slower source. Caching can be implemented at various levels (browser, application server, database, CDN, function level).

Think of it this way: All memoization is caching, but not all caching is memoization. Memoization caches *function results* based on *arguments*, while general caching can store *any data* accessed via a *key* (like a URL, a query string, etc.).

55. Where did you implement caching? Why is it needed?

(Reiterate specific examples from your project, similar to Q38, and explain the "why" for each).

Why it's needed:

- **Performance:** Reduce the time it takes to retrieve data or render content, leading to a faster user experience.
- **Reduced Load:** Decrease the number of requests sent to slower downstream services like databases or external APIs. This can save resources and improve the stability of those services under heavy load.
- **Lower Costs:** Can reduce database server costs or API call costs.
- **Offline Access (Client-side caching):** Browser caching allows users to load some resources even when offline or improves load times by serving from disk cache.

Example from project:

- Caching the list of venues on the homepage: When a user visits the homepage, the list of venues is fetched from the database and stored in Redis. Subsequent visitors within a certain time frame get the list directly from Redis, which is much faster than querying the database every time. This is needed because the list of venues doesn't change very often, and the homepage is likely a high-traffic page.

56. Difference between local storage and cookies.

Both Local Storage and Cookies are ways to store data on the client-side (in the user's browser), but they have significant differences in purpose, capacity, and how they are used.

- **Purpose:**
  - **Cookies:** Primarily for server-side communication, used for session management (storing session IDs), tracking user preferences, and small amounts of user data accessible by the server. Sent automatically with HTTP requests to the originating domain.
  - **Local Storage:** Intended for client-side storage of larger amounts of data that doesn't need to be sent to the server with every request. Useful for offline data, user settings, or caching application state.
- **Capacity:**
  - **Cookies:** Small capacity (typically around 4 KB per cookie, limited number of cookies per domain).
  - **Local Storage:** Larger capacity (typically 5 MB or more per origin/domain).
- **Expiration:**
  - **Cookies:** Can have expiration dates set by the server or client (session cookies expire when the browser closes).
  - **Local Storage:** No built-in expiration; data persists until explicitly cleared by the user or the application.
- **Sent with HTTP Requests:**
  - **Cookies:** Sent automatically with every HTTP request (GET, POST, etc.) to the domain that set them. This is why they are good for session management but also add overhead to requests.
  - **Local Storage: Not** sent automatically with HTTP requests. Data is accessed via JavaScript API (`localStorage.getItem()`, `localStorage.setItem()`). This is good for performance as it doesn't add overhead to network calls.
- **Accessibility:** Both are accessible via JavaScript on the same origin (domain, protocol, port), but cookies are also accessible and managed by the server.

Security Note: Avoid storing sensitive information like passwords or sensitive tokens in Local Storage as it's vulnerable to Cross-Site Scripting (XSS) attacks if your site has XSS flaws. HTTP-only cookies (not accessible via JavaScript) are generally safer for storing session IDs.

Database & SQL

57. What is ON DELETE CASCADE?

`ON DELETE CASCADE` is a referential integrity action that you can specify when defining a foreign key constraint in a relational database. It defines the behavior when a row in the parent table (the table referenced by the foreign key) is deleted.

If a foreign key column in the child table references a primary key column in the parent table, and the foreign key constraint includes `ON DELETE CASCADE`, then when a row is deleted from the parent table, all corresponding rows in the child table that reference the deleted parent row will also be automatically deleted by the database.

Example: If you have a `venues` table and a `shows` table where `shows.venue_id` is a foreign key referencing `venues.id`, and the foreign key constraint has `ON DELETE CASCADE`, deleting a row from the `venues` table will automatically delete all rows in the `shows` table associated with that venue.

This is useful for automatically cleaning up dependent records, but it must be used carefully as it can lead to unintended data loss if not fully understood.

58. How to handle booking more seats than available?

Preventing overbooking requires handling concurrency issues, especially in web applications where multiple users might try to book the same seats simultaneously.

Common approaches:

1. **Database Transactions:** The most reliable method. Wrap the availability check and the booking creation process within a single database transaction.
    a. Start a transaction.
    b. Check the number of available seats for the specific show/section.
    c. If enough seats are available, decrement the available seat count and create the booking record(s).
    d. If not, roll back the transaction.
    e. If successful, commit the transaction. Transactions ensure that either *all* steps succeed or *none* do, preventing partial updates and race conditions.
2. **Row/Table Locking:** More advanced. Acquire a lock on the relevant row (e.g., the row representing the show or seat availability) before checking availability and making the booking. This prevents other transactions from modifying that row until your transaction completes. SQLAlchemy provides ways to acquire locks.
3. **Application-Level Checks with Retries/Queues:** Less ideal for strict consistency, but possible. Perform the check and update, and if it fails due to a race condition (e.g., available

seats becoming zero between check and update), you might retry or inform the user. This is less reliable than database-level solutions for preventing true overbooking.

Using database transactions with a check *inside* the transaction is the standard and most robust way to prevent race conditions leading to overbooking.

59. Why are tickets still showing if the venue is deleted?

If tickets are still showing after a venue is deleted, it implies that the relationship between venues and tickets/shows is not correctly configured to handle deletions, or the application logic doesn't account for it.

Possible reasons:

1. **Missing `ON DELETE CASCADE`:** The foreign key constraint on the `tickets` (or `shows`) table referencing the `venues` table does **not** have `ON DELETE CASCADE` configured. Deleting the venue row leaves the related ticket/show rows orphaned (their `venue_id` foreign key points to a non-existent venue).
2. **`ON DELETE SET NULL` or `ON DELETE RESTRICT`:** The foreign key might be set to `SET NULL` (foreign key becomes null) or `RESTRICT` (deletion is prevented if related records exist).
3. **Application Logic Flaw:** The application code that deletes the venue does not also explicitly delete the related tickets/shows.

Solution: The most common and appropriate fix (depending on desired behavior) is to add `ON DELETE CASCADE` to the foreign key constraint in the database schema definition (or in the ORM model definition which translates to the schema). Alternatively, implement application logic that finds and deletes related tickets/shows *before* deleting the venue. The choice depends on whether you want the database to enforce this automatically (`CASCADE`) or manage it manually in your code.

60. Explain SQL joins, subqueries, aggregate functions, GROUP BY, HAVING.

These are fundamental SQL concepts for querying and manipulating data.

- **Joins:** Combine rows from two or more tables based on a related column between them.
  - `INNER JOIN`: Returns only rows where there is a match in *both* tables.
  - `LEFT JOIN` (or `LEFT OUTER JOIN`): Returns all rows from the left table, and the matched rows from the right table. If no match in the right table, results are `NULL` for right table columns.
  - `RIGHT JOIN` (or `RIGHT OUTER JOIN`): Returns all rows from the right table, and the matched rows from the left table. If no match in the left table, results are `NULL` for left table columns.
  - `FULL JOIN` (or `FULL OUTER JOIN`): Returns all rows when there is a match in *either* left or right table. Results are `NULL` for columns of the table that doesn't have a match.

    - `CROSS JOIN`: Returns the Cartesian product of the two tables (every row from the left combined with every row from the right). Rarely used intentionally except for specific cases.
    - *Example (Inner Join):* `SELECT * FROM orders INNER JOIN customers ON orders.customer_id = customers.id;` (Get orders and customer info for orders that have a customer)
- **Subqueries (Nested Queries):** A `SELECT` query embedded within another SQL statement. They can be used in `WHERE`, `FROM`, or `SELECT` clauses.
    - Used for filtering (e.g., find customers who placed an order), creating derived tables, or selecting single values to be used in the outer query.
    - *Example (Subquery in WHERE):* `SELECT name FROM products WHERE price > (SELECT AVG(price) FROM products);` (Find products more expensive than the average price)
- **Aggregate Functions:** Perform calculations on a set of values and return a single value. Used with `SELECT` clauses.
    - Common functions: `COUNT()`, `SUM()`, `AVG()`, `MIN()`, `MAX()`.
    - *Example:* `SELECT COUNT(*) FROM users;` (Count the total number of users)
    - *Example:* `SELECT AVG(price) FROM products WHERE category = 'Electronics';` (Calculate average price for a category)
- **GROUP BY:** Groups rows that have the same values in one or more columns into a summary row. Aggregate functions are often used with `GROUP BY` to perform calculations for each group.
    - *Example:* `SELECT category, AVG(price) FROM products GROUP BY category;` (Calculate average price *per category*)
- **HAVING:** Used to filter groups created by the `GROUP BY` clause. Similar to `WHERE`, but applied to groups after aggregation, whereas `WHERE` filters individual rows *before* grouping.
    - *Example:* `SELECT category, COUNT(*) FROM products GROUP BY category HAVING COUNT(*) > 10;` (Find categories that have more than 10 products)

61. Why use a separate table for roles/users?

Using separate `users` and `roles` tables with a many-to-many relationship table (e.g., `user_roles`) is a standard database design pattern for implementing RBAC.

Reasons:

- **Normalization:** Avoids storing redundant role information in the `users` table.
- **Flexibility:**
    - Allows a **single user to have multiple roles** easily (many-to-many relationship). Storing roles as a single field (like a comma-separated string) in the user table is bad practice (violates first normal form, difficult to query).
    - Allows a **single role to be assigned to multiple users**.

    - Makes it easy to **add, remove, or rename roles** without altering the user table structure.
- **Maintainability:** Role permissions can be associated with the `Role` table, making it clear what each role is allowed to do.
- **Querying:** Makes querying users by role or roles by user straightforward using joins on the linking table.

A single `role` column in the `users` table (one-to-many relationship, one role per user) is simpler but limits a user to only one role. The separate table structure (many-to-many) is more flexible for complex access control.

62. How would you implement a single table for both users and admins?

You can use a single `users` table to store both regular users and administrators by adding a column that indicates the user's role or status.

Common ways to implement this in a single table:

1. **`is_admin` Boolean Column:** Add a boolean column (e.g., `is_admin`) to the `users` table. `TRUE` indicates an administrator, `FALSE` (or `NULL`) indicates a regular user. This is the simplest method for a binary role system.

SQL
```
CREATE TABLE users (
    id INTEGER PRIMARY KEY,
    username VARCHAR NOT NULL UNIQUE,
    password_hash VARCHAR NOT NULL,
    is_admin BOOLEAN DEFAULT FALSE
);
```

In your application code, you check `user.is_admin`.

2. **`role` String/Enum Column:** Add a string column (e.g., `role`) or use a database ENUM type to store the role name (e.g., 'user', 'admin', 'manager').

SQL
```
CREATE TABLE users (
    id INTEGER PRIMARY KEY,
    username VARCHAR NOT NULL UNIQUE,
    password_hash VARCHAR NOT NULL,
    role VARCHAR DEFAULT 'user' CHECK (role IN ('user', 'admin')) -- Basic
check
);
```

In your application code, you check `user.role`. This is slightly more flexible than a boolean for a few distinct roles but doesn't easily support multiple roles per user.

These single-table approaches are simpler than separate tables for basic role distinctions but lack the flexibility of a normalized many-to-many design for complex RBAC systems.

Frontend & JavaScript

63. What is AJAX?

AJAX stands for Asynchronous JavaScript and XML (although JSON is more commonly used than XML now). It's a set of web development techniques that allows a web page to asynchronously send and retrieve data from a server **without requiring a full page reload**.

This enables creating dynamic and interactive web applications. Instead of the traditional model where user actions trigger a full page submit and reload, AJAX allows JavaScript to make background HTTP requests (using the `XMLHttpRequest` object or the modern `Workspace` API), update parts of the current page with the server's response, and continue interacting with the user.

Example: Clicking a "Like" button updates the like count on the page without refreshing the whole page, or fetching new data when scrolling down an infinite feed.

64. How to access the server without using a browser/URL?

Accessing a server typically implies making a request to a specific network address (IP or domain) and port, which is fundamentally what a browser does when you enter a URL.

However, you can access a server *programmatically* or from non-browser environments without typing a URL into a web browser's address bar. Methods include:

- **Backend-to-Backend Communication:** One server making an HTTP request to another server's API using libraries (like Python's `requests`).
- **Command-Line Tools:** Using tools like `curl` or `wget` from a terminal to make HTTP requests.
- **Desktop or Mobile Applications:** Native applications that make HTTP requests to a backend server API.
- **Background Jobs/Scripts:** Scripts (like Python scripts, cron jobs, Celery tasks) running on a server that perform actions requiring interaction with other services or databases, often via internal network calls or APIs.
- **Specialized Clients:** Using specific client libraries for database access, messaging queues, or other services that communicate over network protocols, but not necessarily HTTP to a "URL" in the browser sense.

So, you're still using network protocols and addresses, but not necessarily through a graphical web browser interface and its URL bar.

65. Why so many Vue files? Difference between components and pages.

In a typical Vue.js project built with the CLI, you often end up with many `.vue` files because the framework encourages a **component-based architecture**.

- **Components:** Are the building blocks of a Vue application. They encapsulate a piece of the UI and its associated logic, styles, and state. Examples: a Button component, a UserCard component, a Modal component, a Header, a Footer. They are designed to be reusable across different parts of the application. A complex page is composed of many smaller components.
- **Pages (or Views):** In the context of Vue Router, "pages" or "views" are often higher-level components that represent an entire route or screen in your application (e.g., `HomePage.vue`, `LoginPage.vue`, `VenueDetails.vue`). A "page" component typically composes multiple smaller, reusable components to build the complete view for that route.

Having many Vue files is a direct result of breaking down the user interface into these smaller, manageable, and reusable components. This improves code organization, maintainability, testability, and collaboration among developers.

66. How did you implement the search function?

(Describe your specific search implementation. A common approach for the MAD project might involve frontend filtering or a backend API.)

Possible Implementations:

1. **Frontend Filtering (for small datasets):**
    a. Fetch all relevant data (e.g., list of venues) upfront.
    b. Use a data property in Vue to hold the search query input (`v-model`).
    c. Use a `computed` property that filters the full data list based on the `searchQuery`.
    d. Render the list using `v-for` on the filtered computed property.
    e. *Pros:* Fast response as no network request needed for each filter.
    f. *Cons:* Not suitable for large datasets (initial load too slow, filtering in JS can become slow).
2. **Backend API Search (for large datasets or complex search):**
    a. Use a data property in Vue for the search query (`v-model`).
    b. Use a `watch` or a method triggered by input/button click to make an API call to the backend.
    c. Send the `searchQuery` as a parameter to a backend search endpoint (e.g., `/api/venues/search?q=query`).
    d. The backend performs the search (e.g., using database queries with `LIKE` or full-text search).
    e. The backend returns the filtered results.
    f. Update a data property in Vue with the `searchResults`.

g. Render the `searchResults` using `v-for`.
h. Implement debouncing/throttling on the input event to avoid making too many API calls while the user is typing.
i. *Pros:* Scalable for large datasets, search logic resides on the backend with access to the database.
j. *Cons:* Requires a network request for each search update (unless debounced).

(Explain which method you chose and why, providing brief code snippets if possible).

67. How to center a login form vertically and horizontally?

You can center an element like a login form both vertically and horizontally within its parent container using CSS layout techniques:

1. **Flexbox:** (Modern and recommended)
    a. Apply `display: flex;` to the parent container.
    b. Apply `justify-content: center;` to horizontally center items.
    c. Apply `align-items: center;` to vertically center items.

CSS

```
.parent-container {
  display: flex;
  justify-content: center; /* Centers horizontally */
  align-items: center;   /* Centers vertically */
  min-height: 100vh; /* Ensure parent is tall enough (e.g., viewport height) */
}
.login-form {
  /* Your form styles */
}
```

2. **CSS Grid:** (Also modern and recommended)
    a. Apply `display: grid;` to the parent container.
    b. Apply `place-items: center;` as a shorthand for `justify-items: center;` and `align-items: center;`.

CSS

```
.parent-container {
  display: grid;
  place-items: center; /* Centers both horizontally and vertically */
  min-height: 100vh;
}
.login-form {
  /* Your form styles */
}
```

3. **Absolute Positioning with Transform:** (Older but still works)
    a. Apply `position: relative;` to the parent container.
    b. Apply `position: absolute;`, `top: 50%;`, `left: 50%;`, and `transform: translate(-50%, -50%);` to the element you want to center (the form).

CSS
```
.parent-container {
  position: relative;
  min-height: 100vh;
}
.login-form {
  position: absolute;
  top: 50%;
  left: 50%;
  transform: translate(-50%, -50%); /* Adjusts position based on element size
*/
  /* Your form styles */
}
```

68. How to move a button in the navbar to the right?

Assuming the navbar is using Flexbox (a common pattern for navbars), you can push an element to the right using `margin-left: auto;`.

If your navbar is a flex container:

HTML
```
<nav style="display: flex; align-items: center;">
  <div class="navbar-left">
    <a href="#">Home</a>
    <a href="#">About</a>
  </div>
  <div class="navbar-right">
    <button>Login</button>
  </div>
</nav>
```

You can apply `margin-left: auto;` to the `.navbar-right` div (or the button directly if it's a direct flex item):

CSS
```
.navbar-right {
  margin-left: auto; /* This pushes this element and subsequent elements to the
```

```
right */
}
```

This works because `margin-left: auto;` consumes all available space on the left side of the element within the flex container.

If not using Flexbox, other methods involve floating the element (`float: right;`) or using absolute positioning, but Flexbox is the most robust and modern approach for navbar layouts.

69. What is a Single Page Application (SPA)? Difference between SPA and multi-page.

- **Single Page Application (SPA):** A web application that loads a single HTML page and dynamically updates the content as the user interacts with it, typically using JavaScript. Navigation within the application (e.g., clicking links) does not trigger a full page reload; instead, JavaScript intercepts navigation, changes the URL using the History API, and renders the new view on the client-side. Frameworks like Vue.js, React, and Angular are commonly used to build SPAs.
- **Multi-Page Application (MPA):** A traditional web application model where each user interaction or navigation action requires the browser to request a new HTML page from the server and perform a full page reload to display the new content.

Key Differences:

- **Page Reloads:** SPA = No full page reloads during navigation. MPA = Full page reloads for every navigation.
- **Performance:** SPA feels faster and smoother after the initial load due to no reloads. Initial load might be slower as more JS is downloaded. MPA has faster initial load (less JS) but slower navigation due to reloads.
- **Architecture:** SPA relies heavily on client-side JavaScript for rendering and routing. MPA rendering is primarily done on the server-side (using templating engines like Jinja2).
- **User Experience:** SPA provides a more fluid, app-like experience. MPA feels more like a traditional website with distinct page transitions.
- **Development Complexity:** SPAs are generally more complex to develop due to client-side routing, state management, and API interactions. MPAs can be simpler for basic sites.
- **SEO:** Historically, MPAs were better for SEO as each page had a distinct URL and the content was readily available to search engine crawlers. Modern SPAs use techniques like Server-Side Rendering (SSR) or pre-rendering to improve SEO.

Project-Specific

70. Why didn’t you let users choose the download location for CSV files?

(This is specific to your implementation decisions. Likely reasons relate to simplicity, security, or relying on browser defaults.)

Likely reasons include:

- **Simplicity/Scope:** Implementing a feature for users to specify a server-side file path based on web input would add significant complexity.
- **Security:** Allowing users to specify arbitrary file paths on the server is a major security risk (Directory Traversal vulnerabilities). The standard and secure way is for the server to generate the file and then send it to the browser with appropriate HTTP headers (`Content-Disposition: attachment; filename="report.csv"`). The browser then handles the download and the user's local file system location.
- **Standard Browser Behavior:** The browser's default download behavior handles where the file is saved on the user's local machine, which is generally sufficient and expected user behavior.

Therefore, it's standard practice for web applications to trigger a download and let the browser/user handle the save location locally, rather than allowing the user to specify a server path via the web interface.

71. What improvements did you make from MAD1 to MAD2?

(Compare the technologies, architecture, and features between the two versions of the project). Common improvements might include:

- **Frontend Framework:** Moving from server-rendered templates (Jinja2 in MAD1) to a client-side SPA framework (Vue.js in MAD2) for a more dynamic UI.
- **API Design:** Developing a clear RESTful API using Flask to separate frontend and backend.
- **Database Management:** Potentially using an ORM (SQLAlchemy) more effectively, improving schema design, or adding features like migrations.
- **Background Tasks:** Introducing Celery for asynchronous processing (emails, reports).
- **Caching:** Implementing caching (Redis) for performance improvements.
- **Security:** Adding or improving authentication (Flask-Login), authorization (RBAC), CSRF protection, proper password hashing.
- **Features:** Adding more complex features like search, reporting, user roles, booking management.
- **Code Structure:** Better modularization using Flask Blueprints, organizing frontend components.

- **Responsiveness & Styling:** Improved UI/UX, better handling of different screen sizes, potentially using a CSS framework.
- **Error Handling:** More robust error logging and user feedback.

(Highlight the most significant technical and functional improvements and explain the value they added).

72. What was the hardest part (frontend/backend)?

(This is subjective. Reflect on your specific challenges).

Possible challenges on the **Frontend**:

- State management (especially in a growing SPA).
- Handling complex component interactions and communication (props, events, slots).
- Implementing complex UI logic or animations.
- Ensuring responsiveness across many devices.
- Debugging asynchronous operations and data flow.
- Integrating with backend APIs and handling different response states.

Possible challenges on the **Backend**:

- Database design and optimizing complex queries.
- Handling concurrency and race conditions (e.g., during booking).
- Implementing robust security measures (authentication, authorization, preventing attacks).
- Designing a clean and scalable API.
- Setting up and managing background task queues (Celery, Redis).
- Handling file uploads or report generation.
- Deployment and environment configuration.

(Choose the area where you faced the most significant technical hurdles and explain what the challenge was and how you overcame it).

73. What other features can be added?

(Brainstorm features relevant to a ticketing/venue booking application). Potential features include:

- **Payment Gateway Integration:** Allow users to purchase tickets online.
- **Seat Selection:** A graphical interface for users to select specific seats on a seating map.
- **User Profiles:** Allow users to view past bookings, saved venues, etc.
- **Reviews and Ratings:** Users can leave reviews for venues or shows.
- **Notifications:** Email or in-app notifications for booking confirmations, reminders, or new shows.

- **Wishlist/Following:** Users can mark venues as favorites or follow venues/artists.
- **Recommendation Engine:** Suggest shows or venues based on user history or preferences.
- **Search Filters and Sorting:** More advanced options for finding shows/venues (by date, genre, price, location).
- **Admin Dashboard:** Analytics, user management, content management system for venues/shows.
- **Promotions/Discounts:** Implement discount codes or special offers.
- **QR Code Tickets:** Generate scannable QR codes for tickets.
- **Internationalization (i18n) & Localization (l10n):** Support for multiple languages and regions.

74. How would you implement a "favorite theatre" feature?

Implementing a "favorite theatre" feature involves both backend and frontend work:

Backend Implementation:

1. **Database Schema:**
    a. Create a new table, e.g., `user_favorites`.
    b. This table would typically have two foreign keys: `user_id` referencing the `users` table and `venue_id` referencing the `venues` table.
    c. Create a composite primary key or unique constraint on `(user_id, venue_id)` to prevent a user from favoriting the same venue multiple times.

SQL
```
CREATE TABLE user_favorites (
    user_id INTEGER NOT NULL,
    venue_id INTEGER NOT NULL,
    PRIMARY KEY (user_id, venue_id), -- Prevent duplicate favorites
    FOREIGN KEY (user_id) REFERENCES users(id) ON DELETE CASCADE, -- If user
deleted, remove their favorites
    FOREIGN KEY (venue_id) REFERENCES venues(id) ON DELETE CASCADE -- If venue
deleted, remove from favorites
);
```

2. **API Endpoints:**
    a. `POST /api/users/<user_id>/favorites` (or `/api/favorites` with user ID from token): Endpoint to add a venue to favorites. Request body contains `venue_id`.
    b. `DELETE /api/users/<user_id>/favorites/<venue_id>` (or `/api/favorites/<venue_id>`): Endpoint to remove a venue from favorites.
    c. `GET /api/users/<user_id>/favorites` (or `/api/favorites`): Endpoint to retrieve a list of the user's favorited venue IDs or full venue objects.
    d. `GET /api/users/<user_id>/favorites/status/<venue_id>`: Optional endpoint to check if a specific venue is favorited by the user.

Frontend Implementation (Vue.js):

1. **UI Element:** On the venue details page or venue list item, add a button or icon (e.g., a star).
2. **State:** In the component, maintain a boolean data property, e.g., `isFavorited`, initialized based on whether the current user favorites this venue.
3. **Initial Check:** When the venue component loads, make an API call (e.g., `GET /api/users/me/favorites/status/<venue_id>`) to determine the initial state of `isFavorited`.
4. **Click Handler:** Attach a click event listener to the star/button.
    a. When clicked, check the current `isFavorited` state.
    b. If `isFavorited` is false (want to favorite): Make a `POST` request to the "add favorite" endpoint.
    c. If `isFavorited` is true (want to unfavorite): Make a `DELETE` request to the "remove favorite" endpoint.
    d. Update the `isFavorited` state based on the successful API response.
    e. Update the appearance of the star/button.
5. **Display Favorites:** Create a separate "Favorites" page or section that makes a `GET` request to the "get favorites" endpoint and displays the list of favorited venues.

This approach separates concerns, uses RESTful principles for the API, and manages the state effectively on the frontend.

75. What technologies did you use in the app?

(List the main technologies from all parts of the stack). Based on the question topics, the likely stack is:

- **Frontend:** Vue.js (with Vue Router), JavaScript, HTML5, CSS3.
- **Backend:** Flask (Python), Flask-SQLAlchemy (for database interaction), Werkzeug (for security utilities like hashing), Flask-Login (for authentication), Flask-WTF (for forms and CSRF).
- **Database:** SQLite (development), potentially PostgreSQL or MySQL (production/demonstration).
- **Task Queue:** Celery.
- **Message Broker/Cache:** Redis.
- **Other Libraries:** `requests` or `axios` (for making HTTP requests), password hashing library (like `bcrypt` if not using Werkzeug's default).

76. What are your thoughts on scaling the application?

Scaling a web application involves designing it to handle increased load (more users, more data, more requests) without performance degradation.

Thoughts on scaling this application:

- **Horizontal Scaling:** The current architecture (Flask for backend, Celery for tasks, Redis for cache/broker, relational DB) is reasonably well-suited for horizontal scaling. You can run multiple instances of the Flask application, multiple Celery workers, and potentially multiple Redis instances.
- **Stateless Backend:** The Flask application instances should be stateless, meaning user sessions should be managed externally (e.g., in Redis or a database) rather than in the memory of a single Flask process. This allows requests from the same user to be handled by any available Flask instance behind a load balancer.
- **Database Scaling:** The relational database (SQLAlchemy) will likely become a bottleneck under heavy read/write load. Scaling options include:
    - **Vertical Scaling:** Upgrade the database server hardware (CPU, RAM, faster storage).
    - **Read Replicas:** Create read-only copies of the database to distribute read traffic.
    - **Sharding/Partitioning:** Splitting data across multiple database servers based on criteria (e.g., user ID ranges, geographical location). This is more complex.
    - **Caching:** Heavily leverage caching (Redis) to reduce database reads.
- **Task Queue Scaling:** Celery workers can be scaled up or down based on the volume of background tasks.
- **Caching Scaling:** Redis can be scaled using clustering or replication.
- **Frontend Scaling:** Vue.js itself scales well; performance bottlenecks are usually related to complex components, excessive rendering, or inefficient API calls. Lazy loading routes and components helps with initial load time.
- **Load Balancing:** A load balancer is essential to distribute incoming requests across multiple instances of the Flask application and potentially multiple Redis/database replicas.
- **Monitoring & Profiling:** Crucial for identifying performance bottlenecks as the application scales.

The current architecture provides a good foundation, but scaling to very high traffic levels would require careful planning, optimization, and potentially transitioning to more distributed or specialized database solutions and infrastructure management.

General & Personal

77. What are you doing besides this degree? Why are you pursuing it?

(This is a personal question. Answer honestly based on your own experiences and motivations.)

Example Answer Structure:

- Mention significant activities outside academics (e.g., job, internships, personal projects, volunteering, clubs, hobbies).
- Explain your motivation for pursuing the degree (e.g., passion for technology, career goals, specific interest in the field, intellectual curiosity, skill development).

78. What do you understand by web application development?

Web application development is the process of creating software applications that run on web servers and are accessed by users over the internet using a web browser.

It typically involves:

- **Client-side (Frontend) Development:** Building the user interface and user experience that runs in the user's browser (using HTML, CSS, JavaScript, and frameworks like Vue.js). This includes rendering content, handling user input, and communicating with the backend.
- **Server-side (Backend) Development:** Building the core logic, APIs, database interactions, authentication, and business rules that run on the server (using languages like Python/Flask, Node.js/Express, etc.).
- **Database Management:** Designing, implementing, and maintaining databases to store application data.
- **API Development:** Creating interfaces (APIs) that allow the frontend and other services to communicate with the backend.
- **Deployment and Operations (DevOps):** Packaging, deploying, and managing the application on servers or cloud platforms.
- **Security:** Ensuring the application is protected against common web vulnerabilities.

It's a multi-disciplinary field requiring knowledge across these areas to build functional, scalable, and secure applications delivered over the web.

79. What frameworks have you used in your project?

(List the main frameworks used):

- **Backend Framework:** Flask (Python)
- **Frontend Framework:** Vue.js (JavaScript)
- Potentially mention related ORM framework like SQLAlchemy (though often considered a toolkit/library rather than a full framework in this context, it's worth mentioning its significant role).

80. Do you prefer frontend or backend?

(This is a subjective question. Answer honestly, explaining why you lean towards one or the other or if you enjoy both.)

Possible answers:

- **Prefer Frontend:** Enjoy the visual aspect, building user interfaces, immediate feedback, working with design, focusing on user experience.

- **Prefer Backend:** Enjoy the logic, data modeling, system design, performance optimization, working with databases and APIs, solving complex problems behind the scenes.
- **Enjoy Both (Full-stack):** Appreciate understanding the entire stack, seeing how frontend and backend connect, the challenge of working across layers.

Explain your preference with a specific reason or examples from your project that highlight why you enjoyed working on that part.

81. What is your confidence level in JavaScript/frontend technologies?

(Provide an honest self-assessment. Be positive but realistic. Use a scale or descriptive terms.)

Example answers:

- **On a scale of 1 to 5, I'd say a 4.** I'm confident in building functional UIs, working with Vue.js components, handling API calls, and implementing common frontend patterns. I'm always learning and exploring more advanced topics like performance optimization or complex state management.
- **I'm quite confident.** I enjoy working with JavaScript and feel comfortable building interactive user interfaces with Vue. I understand core concepts like reactivity, components, and routing. There's always more to learn, but I'm capable of tackling new challenges in frontend development.
- **I'm proficient.** I have a solid understanding of JavaScript fundamentals and experience using Vue.js to build single-page applications. I can design components, manage state, and integrate with backend services.