ChatGPT

# Microsoft Event Page Structure Analysis

## HTML Layout and Structure

Each Microsoft event page is structured as a standalone HTML document with a clear division of content and a registration form. The page typically includes the global Microsoft header and footer (for consistent branding) and a main content area dedicated to the event details. The main content is often split into two columns: **event information** on the left and the **registration form** on the right (on mobile screens these stack vertically for responsiveness).

In the HTML snippet below (e.g. `index.html`), you can see the overall layout. The left column contains details like event title, date/time, location, and description, while the right column contains the sign-up form fields (First Name, Last Name, Email, Phone, Job Role, Company, Country, etc.) [1]. This matches the fields that the underlying system expects (e.g. `me_firstname`, `me_lastname`, `me_email`, `me_phonenumber`, `me_companyname`, `me_country`, etc. in the event registration database) [1]. An example HTML structure might look like:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Supercharge Data Security with Copilot – Microsoft Event</title>
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <!-- External CSS link (if using separate stylesheet) -->
  <link rel="stylesheet" href="css/styles.css">
</head>
<body>
  <!-- Microsoft global header (navigation bar) -->
  <header>
    <!-- ... global Microsoft site navigation HTML ... -->
  </header>

  <!-- Main Event Content -->
  <main>
    <section class="event-hero">
      <!-- Left column: Event details -->
      <div class="event-info">
        <span class="event-format">Digital</span>
        <h1 class="event-title">Supercharge Data Security & Compliance with
Security Copilot in Microsoft Purview</h1>
        <p class="event-location"><strong>Where:</strong> Online</p>
        <p class="event-datetime"><strong>When:</strong> Tuesday, August 12,
```

```
2025, 12:00 – 1:00 PM (GMT+08:00)</p>
        <p class="event-timezone">*Localized times:* Singapore – 12:00 PM SGT;
Sydney – 2:00 PM AEST; Mumbai – 9:30 AM IST</p>
        <p class="event-language"><strong>Delivery language:</strong> English</
p>
        <p class="event-caption"><strong>Closed captioning:</strong> English</p>
        <p class="event-description">
          As organizations embrace generative AI, the stakes for data security
and compliance have never been higher.
          Join us for an exclusive briefing to explore how Microsoft is
transforming data security and compliance teams' experience
          with the power of Copilot for Security — natively integrated into
Microsoft Purview.
        </p>
      </div>

      <!-- Right column: Registration form -->
      <div class="registration-form">
        <h2>Register Now</h2>
        <form id="registerForm">
          <div class="form-group">
            <label for="firstName">First Name</label>
            <input id="firstName" name="firstName" type="text" required>
          </div>
          <div class="form-group">
            <label for="lastName">Last Name</label>
            <input id="lastName" name="lastName" type="text" required>
          </div>
          <div class="form-group">
            <label for="email">Email Address</label>
            <input id="email" name="email" type="email" required>
          </div>
          <div class="form-group">
            <label for="phone">Phone Number</label>
            <div class="phone-input">
              <!-- Country code dropdown -->
              <select id="phoneCountry" name="phoneCountry" required>
                <option value="+1">+1 (US)</option>
                <option value="+44">+44 (UK)</option>
                <!-- ...other country codes... -->
              </select>
              <input id="phoneNumber" name="phoneNumber" type="tel" required
placeholder="Phone Number">
            </div>
          </div>
          <div class="form-group">
            <label for="jobRole">Job Role</label>
            <input id="jobRole" name="jobRole" type="text" placeholder="e.g. IT
```

```
Administrator">
          </div>
          <div class="form-group">
            <label for="company">Company Name</label>
            <input id="company" name="company" type="text" required>
          </div>
          <div class="form-group">
            <label for="country">Country/Region</label>
            <select id="country" name="country" required>
              <option value="">-- Select Country --</option>
              <option value="US">United States</option>
              <option value="CA">Canada</option>
              <option value="SG">Singapore</option>
              <!-- ...all country options... -->
            </select>
          </div>
          <!-- Privacy consent and submit -->
          <div class="form-group">
            <input id="acceptTerms" name="acceptTerms" type="checkbox" required>
            <label for="acceptTerms">I agree to the Microsoft Privacy
Statement</label>
          </div>
          <button type="submit" id="registerButton">Register</button>
        </form>
      </div>
    </section>
  </main>

  <!-- Microsoft global footer -->
  <footer>
    <!-- ... global footer content (links, copyright) ... -->
  </footer>

  <!-- Scripts -->
  <script src="js/scripts.js"></script>
</body>
</html>
```

In the code above, the structure is organized for clarity: a `<section class="event-hero">` wraps the two columns. The left column (`.event-info`) contains text content, and the right column (`.registration-form`) contains the form. Each input is labeled and grouped in a `.form-group` for styling purposes. Notice the phone number field includes a country code dropdown plus the phone input – this matches the actual page's approach of collecting country code and number separately [2]. The form also includes a checkbox for agreeing to the Privacy Statement, which is common on Microsoft event registration pages. After the form, the page includes the standard Microsoft footer.

## External and Inline CSS Styles

The styling for the event page is handled by a combination of external CSS and possibly some inline styles for dynamic elements (like an inline style on a container to set a background image). Typically, a main stylesheet (e.g. `css/styles.css`) defines the layout, typography, and color scheme. Microsoft's design language uses a clean, modern look – likely leveraging web-safe fonts (e.g. Segoe UI, the standard Microsoft font) and a responsive grid or flexbox layout for the two columns. The page may use a CSS framework behind the scenes; for example, older event pages often used Bootstrap for grid and form styling (class names like `.form-group` suggest a Bootstrap-like structure). It's also possible Microsoft uses a custom in-house style library or Fluent UI principles for consistency.

Below is a sample of the CSS that could be used to achieve the layout and style shown in the HTML. It includes responsive design, basic typography, and form styling:

```
/* Container and layout */
.event-hero {
  display: flex;
  align-items: flex-start;
  justify-content: space-between;
  background: url("../images/event-banner.jpg") center/cover no-repeat;
  padding: 2rem;
  gap: 2rem;
}
.event-info, .registration-form {
  background: rgba(255,255,255,0.8); /* light overlay to ensure text legibility
over image */
  padding: 1.5rem;
  border-radius: 4px;
  flex: 1;
}
/* To make the left column larger than right, one could adjust flex ratios if
needed */
.event-info {
  flex: 2;
}
.registration-form {
  flex: 1;
}

/* Typography */
body {
  font-family: "Segoe UI", Arial, sans-serif;
  color: #1b1b1b;
  margin: 0;
  background: #f8f8f8;
}
```

```
.event-title {
  font-size: 1.5rem;
  margin: 0.5rem 0;
}
.event-format {
  display: inline-block;
  font-size: 0.9rem;
  font-weight: bold;
  padding: 0.2em 0.5em;
  background: #f3f2f1;  /* light gray pill for "Digital"/"In-person" label */
  border-radius: 4px;
  margin-bottom: 1rem;
}
.event-location, .event-datetime, .event-language, .event-caption, .event-
description {
  margin: 0.25rem 0;
}
.event-description {
  margin-top: 1rem;
}

/* Form styles */
.registration-form h2 {
  font-size: 1.25rem;
  margin-bottom: 1rem;
}
.form-group {
  margin-bottom: 1rem;
}
.form-group label {
  display: block;
  font-weight: 600;
  margin-bottom: 0.3rem;
}
.form-group input,
.form-group select {
  width: 100%;
  padding: 0.5rem;
  font: inherit;
  border: 1px solid #ccc;
  border-radius: 3px;
}
.phone-input {
  display: flex;
}
.phone-input select {
  max-width: 30%;
  margin-right: 0.5rem;
```

```
}
button#registerButton {
  background: #0078D4; /* Microsoft blue */
  color: #fff;
  padding: 0.6rem 1.2rem;
  font-size: 1rem;
  border: none;
  border-radius: 3px;
  cursor: pointer;
}
button#registerButton:hover {
  background: #005EA6;
}
/* Responsive adjustments for narrower screens */
@media(max-width: 768px) {
  .event-hero {
    flex-direction: column;
  }
  .event-info, .registration-form {
    width: 100%;
    margin-bottom: 2rem;
  }
}
```

In this CSS, `.event-hero` is given a background image (`event-banner.jpg`) which would be the banner image for the event (often a relevant stock photo or graphic provided for the event). The CSS uses a translucent white background on `.event-info` and `.registration-form` (`rgba(255,255,255,0.8)`) to improve text contrast against the background image. The layout uses flexbox to place the two columns side by side, and a media query ensures that on small screens they stack vertically. Form controls are styled with basic padding and borders; the **Register** button is styled in Microsoft's signature blue color for prominence.

If the actual Microsoft event page uses an existing CSS framework, many of these styles might come from that framework's classes (for example, if Bootstrap were used, classes like `.row`, `.col-md-6`, etc., would handle the layout, and default form-control styling would apply to `<input>` elements). In our breakdown, we show custom CSS for clarity and easy customization.

**Inline Styles:** In the actual page, dynamic content like the event banner image might be set via inline style or through CSS loaded at runtime. For example, the hero section's background might be inserted with an inline style attribute (e.g., `<section class="event-hero" style="background-image:url('.../banner.jpg');">` if each event has a unique image). Other than that, most styling is likely kept in external CSS for reusability. We haven't shown inline styles above except what might logically be present. The page does not heavily rely on inline styles aside from the possible background image or minor conditional styling.

## JavaScript Files and Logic

The event page is dynamic, meaning it likely loads event data and handles form submission using JavaScript. The scripting is likely contained in one or more JS files – often minified in production. It's common that Microsoft's event pages function as a single-page application (SPA) or use client-side scripting to fetch data from an API and to submit registrations. Under the hood, the event data is stored in a database (Microsoft uses Dataverse/Dynamics 365 for events management, as evidenced by OData API calls) [3].

**File Structure:** In a project setup, we might have a file like `js/scripts.js` (or multiple files/bundles) handling the logic. For clarity, here's an outline of what the JavaScript does:

1. **Fetch Event Details:** On page load, a script will fetch the event's details (title, description, date/time, etc.) from an API endpoint. In the actual site, this is done via a call to `https://msevents.microsoft.com/api/GetEvents?eventid=<ID>` which returns a JSON object with all the public info about the event [3]. This data is then used to populate the HTML (if server-side didn't already render it). In our static example above, we hard-coded the values into HTML for simplicity, but on Microsoft's site this likely happens dynamically through JS if the page is an SPA. The use of an API suggests the page might be built with a framework like **React** or **Angular**, where the initial HTML is just a container and the content is injected after fetching. However, it could also be a server-rendered page that already includes the data. The presence of API calls in the vulnerability report [3] suggests a client-side dynamic approach.

2. **Form Validation:** The JavaScript likely performs validation on the form. This could include checking that required fields are not empty, email is in proper format, and phone number meets criteria. Some validation might rely on HTML5 (e.g. the `required` attribute and input types), and additional custom checks can be done in JS.

3. **Form Submission (Registration Logic):** When the user submits the form, the script will typically make a POST request to a registration endpoint. In the actual Microsoft Events system, this might be an API like `/api/RegisterEvent` or a similar endpoint that creates a new registration record. Before creating a registration, the site might call a **"check registration"** endpoint (`/api/CheckEventRegistration`) to see if the email is already registered for the event [3] [1] – this is used to prevent duplicate sign-ups (the vulnerability report shows this endpoint was used to verify emails and was exploited via OData injection). If the email is not yet registered, the script proceeds to submit the new registration. After a successful submission, the user might see a confirmation message or receive a confirmation email with event details (often a join link for virtual events). If the email is already registered, the page might display an error or a note that the user is already signed up.

4. **Libraries and Frameworks:** The logic for these actions could be written in plain JavaScript, but it's likely part of a larger framework. For example, if built on **React**, the page might have React components for the form and use the Fetch API or Axios to call the endpoints. If it's a **Power Pages (Dynamics 365 Portal)** implementation, it might use jQuery for AJAX calls and some Dynamics portal scripts for form submission. We don't see explicit references in the HTML to script libraries like `<script src="https://.../react.js">` because in production these are bundled. However,

one can infer from the dynamic behavior that a modern JS framework is in play. There is no evidence of heavy use of jQuery on these pages, so a framework-driven approach (React/Angular or even a lightweight Vanilla JS) is plausible.

Here's a pseudo-code example (in a simplified way) of what the `scripts.js` might contain to handle data loading and form submission:

```
// scripts.js

document.addEventListener('DOMContentLoaded', function() {
  // Fetch event details via API (if needed)
  const eventId = new URL(window.location.href).searchParams.get('id');
  if (eventId) {
    fetch(`/api/GetEvents?eventid=${eventId}`)
      .then(response => response.json())
      .then(data => {
        // Populate the page with event data
        document.querySelector('.event-title').textContent = data.title;
        document.querySelector('.event-datetime').textContent =
formatEventDateTime(data);
        // ... set other fields like description, location, etc.
      })
      .catch(err => console.error('Error loading event data', err));
  }

  // Form submission handler
  const form = document.getElementById('registerForm');
  form.addEventListener('submit', function(e) {
    e.preventDefault();
    // Basic validation (additional to HTML5 constraints)
    if (!form.checkValidity()) {
      // If any field is invalid according to HTML5, the browser will block
submission.
      return;
    }
    // Construct registration payload from form fields
    const registrationData = {
      firstName: form.firstName.value,
      lastName: form.lastName.value,
      email: form.email.value,
      phoneNumber: form.phoneCountry.value + form.phoneNumber.value,
      jobRole: form.jobRole.value,
      company: form.company.value,
      country: form.country.value
    };

    // Optional: Check if already registered
```

```
    fetch('/api/CheckEventRegistration', {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify({ email: registrationData.email, eventId: eventId })
    })
    .then(res => res.json())
    .then(result => {
      if (result.exists) {
        alert("You are already registered for this event.");
      } else {
        // Proceed to register
        return fetch('/api/RegisterEvent', {  // fictitious endpoint for example
          method: 'POST',
          headers: { 'Content-Type': 'application/json' },
          body: JSON.stringify(registrationData)
        });
      }
    })
    .then(res => {
      if (res && res.ok) {
        // Show confirmation (could redirect to a thank-you page or display a
message)
        form.innerHTML = "<p>Thank you for registering! You will receive a
confirmation email shortly.</p>";
      } else if(res) {
        // Handle server-side validation errors
        alert("Registration failed: " + res.statusText);
      }
    })
    .catch(err => console.error('Error during registration', err));
  });
});

// Helper to format date/time from data (implementation would parse the event
time info)
function formatEventDateTime(data) {
  // e.g., combine date and time fields from data into a string
  return data.date + " " + data.startTime + " – " + data.endTime + " (" +
data.timeZone + ")";
}
```

In the above pseudo-code, we show the flow: get the `eventId` from the URL, fetch the event details via `GetEvents` API (the actual JSON structure would have fields like title, description, start/end times, time zone, etc.). Then the script populates the DOM elements with this data. The form submission code gathers all input values and sends them to a `RegisterEvent` API (the actual endpoint name could differ – it might be an OData endpoint or a custom API in Microsoft's system). Before that, it uses `CheckEventRegistration` to illustrate how duplicate-check might be done (the vulnerability write-up

confirms such an API exists [3]). If the email is new, it submits the data; if not, it alerts the user they're already registered. After successful registration, it provides feedback to the user.

**Obfuscation/Minification:** The actual `scripts.js` delivered by Microsoft is likely minified or bundled (all scripts in one). In a production environment, you might see a file name like `main.<hash>.js` loaded, containing all the JavaScript code in a compressed form. The logic we outlined would be inside that bundle. To recreate or adapt this, you don't need the minified code verbatim – you can write your own scripts (like the example) to perform similar functions. If you were to use a framework (React, Angular, etc.), the code structure would be different (split into components or services), but the end effect is the same: fetch data, render page, handle form submission via API calls.

## Images, Fonts, and Other Assets

The event page uses a few asset types:

- **Images:** Typically, each event has a banner image. In our example, `event-banner.jpg` is the hero background image shown behind the event info. The actual site likely stores these images in a content database or CDN. For instance, the banner might be loaded from a Microsoft storage location or embedded via CSS. Additionally, the Microsoft logo (in the header) and possibly other icons (e.g., a favicon, social media icons) are used. Those are usually referenced via CDN links (for example, the Microsoft logo might come from `microsoft.com` global assets). If you recreate this page, you can include your own images (in an `/images` folder as shown in our structure). Ensure to style the image (or background) to be responsive. In our CSS, we used a background image with `cover` to automatically resize/crop for different screens.

- **Fonts:** Microsoft's web pages generally use **Segoe UI** as the primary font (which is a standard font on Windows and available on many systems). They might also use fallback fonts like Arial or sans-serif for non-Windows systems. In some cases, Microsoft might include a reference to a webfont or use system fonts. In our CSS, we set `font-family: "Segoe UI", Arial, sans-serif;` to match the look. If you need the exact Segoe UI look on all platforms, you might use the Segoe UI Variable font via an @font-face or a link to a Microsoft CDN, but usually it's not necessary if you have good fallbacks.

- **Icons/Graphics:** The page might include small icons (for example, a calendar icon next to the date, or a location pin next to the location text). The provided HTML didn't explicitly show such icons, but these could be included via icon fonts or inline SVG. If using Microsoft's design kit, you might see Segoe Fluent Icons or similar. For simplicity, our example omits them, but you can always add an `<img>` or `<span class="icon icon-calendar"></span>` with appropriate CSS if desired.

- **Other assets:** The global header and footer might load some script or CSS for functionality (for example, the header's dropdown menus or search box). These are part of Microsoft's global assets. When recreating the page for personal use, you likely won't need those scripts unless you plan to replicate the header exactly. If you do include the header/footer, note that Microsoft often provides an embed script to load them. Otherwise, you can simplify by creating your own static header.

The file/folder structure for assets in our breakdown is:

```
/project-root
├── index.html
├── css/
│   └── styles.css
├── js/
│   └── scripts.js
└── images/
    ├── event-banner.jpg
    └── logo.png   (for example, Microsoft logo or your own logo)
```

All links in the HTML (CSS, JS, images) are relative to this structure. In a real scenario, the CSS and JS might be served from a CDN or built pipeline, but this structure makes it easy to manage and edit components.

## Frameworks and Libraries Used

From the analysis, the Microsoft event page appears to leverage modern web frameworks and possibly some Microsoft-specific libraries:

- **Front-end Framework:** It is likely a React-based application. Microsoft has often used React for many of its web portals. The fact that content is fetched via an API and dynamically injected [3] hints at a single-page application or at least heavy client-side scripting. If it were a server-rendered page, the need to call `GetEvents` from the browser would be less (the server could inject the data). So, frameworks like React (with perhaps Redux for state management) or Angular could be at play. Without the un-minified source, we deduce this from typical practices and the structure of the data calls.

- **UI Libraries/Styles:** If using React, Microsoft might have used Fluent UI (formerly Office UI Fabric) for styling components, which provides React components for forms, layout, etc. However, the form markup we reconstructed looks fairly generic, not showing obvious Fluent UI components (which would have different class names and DOM structure). It's possible they used a simpler approach or custom components. Another possibility is the use of **Bootstrap** or a grid system for quick layout, especially if this is a Power Platform Portal – older Dynamics 365 portals did use Bootstrap 3 by default. The class names `.form-group` and general layout could indeed be Bootstrap conventions. If so, the page would include references to Bootstrap's CSS and JS (and jQuery, since Bootstrap 3/4 requires it). We haven't seen those references directly, but it's something to consider. In a personal adaptation, you can choose either to use Bootstrap (to get similar form styling out-of-the-box) or just stick with custom CSS as shown.

- **Microsoft-specific Scripts:** The global header/footer might be injected by a Microsoft script (for example, some Microsoft sites use a script from `navclient.microsoft.com` to populate the menu). This is not strictly part of the event page content, but part of the overall page template. We mention it for completeness – it loads the top navigation menu items, search, etc., so the HTML in the header/footer might not all be hard-coded. Since this is not crucial to replicating the visual layout, you can either exclude it or replace it with your own navigation.

- **Dataverse and Power Platform:** The backend of this events site is likely powered by Dynamics 365 Marketing or a similar events-management solution. The use of OData endpoints (observed in the security analysis) indicates Microsoft Dataverse (OData is the API for Dataverse) [3]. While this is behind the scenes, it does mean the front-end had to interact with these endpoints using AJAX/Fetch calls. To recreate functionality, you don't need Dataverse – you could use any backend or simply email the form results – but it's useful to know what frameworks were involved on Microsoft's side. If one were trying to build a similar system, they might use Microsoft Power Pages (which comes with a lot of this functionality out-of-the-box, including the event registration tables and forms). For our purposes (a static/dynamic page for personal use), you can manage without those heavy frameworks by handling form submission in a way that suits your project (for example, sending the data to an email or storing in a simple database).

## Recreating and Adapting the Page

Because the original site's source files are bundled/minified, we have provided clean, unminified examples above that capture the structure and style. If you view the real page's source, you'd mostly see references to scripts and a few root HTML elements rather than the full content (since content is injected via scripts). Our breakdown serves as a developer-friendly template.

To adapt this for your own project:

- **HTML:** Use `index.html` as the template. Replace the event-specific text (titles, dates, description, etc.) with your content. Keep the form structure the same to maintain the layout. If you don't need certain fields, you can remove them (just ensure to adjust any related validation in the script). If you need additional fields, you can copy the pattern of the form-group.

- **CSS:** You can take `styles.css` and tweak the colors, spacing, and font as desired. The provided CSS is commented for clarity. For instance, if you have a different brand color for the button, change the background on `#registerButton`. The layout CSS will handle responsiveness; you can test it by resizing the browser.

- **JS:** The `scripts.js` example can be implemented in plain JavaScript as given. If your adaptation doesn't need to actually submit to a server (e.g., maybe you'll just collect responses via email or a third-party form service), you can simplify or remove the submission logic. On the other hand, if you want similar functionality, you'll need to create an endpoint on your server to handle the POST and store the data (or integrate with an API of your choice). The example shows how you might integrate the logic. If not using an API, you could even remove the fetch calls and simply display a "Thank you" message on submit for a demo page.

- **Minification/Bundling:** For a personal project, it's not strictly necessary to bundle or minify the files, but in production it's good practice. You can use tools like Webpack or parcel to bundle your JS (especially if you break it into modules or use a framework). The same goes for CSS (you might use SASS/SCSS and then compile and minify). However, if the page is simple, you can also just use the files as-is. The structure we provided is straightforward to edit in any code editor and does not require a build step unless you choose to.

By following this breakdown, you should be able to recreate a page very similar in appearance and behavior to the Microsoft event page in question. All key components – HTML layout, CSS styling, JS logic, and assets – have been covered. As an educational exercise, this shows how a modern event registration page is put together, without copying any proprietary code (we've written the samples ourselves based on observed structure). You can confidently use this as a starting point for your own event page, customizing as needed for your content and ensuring you're not using any Microsoft copyrighted assets beyond the layout concepts (which is within fair use for learning).

**References:**

- Microsoft's event page uses a data-driven approach (via OData APIs) to populate event info [3], and the form fields correspond to fields like first name, last name, phone, company, country, etc., as confirmed by a security analysis of the backend [1]. These insights guided the reconstruction of the form structure and logic.
- The dynamic nature of content loading and registration was inferred from the same analysis, which showed how the event data and registrations are handled through API calls [3]. This ensures our adapted code aligns with the real page's functionality, even though we implement it in a simplified manner for personal use.

---

[1] [3] Microsoft Events Leak, Part I: Leaking Event Registration and Waitlist Databases | blog.faav.net
https://blog.faav.net/microsoft-events-leak-part-i

[2] Microsoft Azure Code Smarter, Faster: Empowering India's ...
https://www.instagram.com/reel/DK7NdPDTV_G/