ChatGPT

CyberApp_FIA Project Code Walkthrough

Welcome_Page.aspx (Landing Page)

This is the public landing page that introduces the application and directs users to either sign up or log in. The ASPX markup defines a welcoming interface with some styling and images, but functionally its main elements are two buttons: **"Create Account"** and **"Sign In."** These are implemented as ASP.NET Button controls that navigate to the appropriate pages. For example, the **Create Account** button's markup sets a `PostBackUrl` to the sign-up page (`~/Account/CreateAccountPage.aspx`) and an `OnClick` handler `BtnCreate_Click`. (In the code-behind, `BtnCreate_Click` is currently empty – the navigation is handled by the `PostBackUrl` itself.) The **Sign In** button similarly points to the login page (`~/Account/Login.aspx`). Apart from navigation, the welcome page's code-behind (`Welcome_Page.aspx.cs`) does not contain any logic (the `Page_Load` is empty). In summary, the Welcome page serves as a static entry point with branding and links to the account pages.

CreateAccountPage.aspx (User Registration)

The Create Account page allows new users to register (intended for Participants). The ASPX form includes fields for **First Name**, **Last Name**, **Email**, **Password**, **Confirm Password**, and a consent checkbox (agreeing to terms). ASP.NET validation controls are used for required fields and to ensure the email is in a valid format and the passwords match. Notably, there is an ASP HiddenField `Role` with value "Participant" – meaning all accounts created here default to the Participant role (the interface does not allow selecting other roles).

In the code-behind (`CreateAccountPage.aspx.cs`), the `BtnSignUp_Click` handler implements the registration logic. It first validates the form input on the server side: if any validators failed (`Page.IsValid` is false), it returns immediately. It also double-checks that the consent checkbox is checked; if not, it sets an error message on a `FormMessage` label and stops processing. Once validation passes, the code ensures the user data XML exists by calling `EnsureXml()`. This method creates the `App_Data/users.xml` file on first run if it doesn't already exist. It writes a minimal XML structure `<users version="1"></users>` as the root if needed.

After ensuring the file, the code enforces **unique email** registration. It calls a helper `EmailExists(email)` which loads the users XML and uses an XPath query to find any user with a matching email (case-insensitive). The XPath uses `translate()` to normalize case. If a user with that email is found, the sign-up is aborted with a friendly error message ("That email is already registered.").

Next is password handling. The code generates a new **unique user ID** (`id`) and a salt for the password. The user ID is generated as a GUID string (`Guid.NewGuid().ToString("N")`), which will be stored as the user's `id` attribute. For the password, a cryptographic random salt is created (using `RNGCryptoServiceProvider`) and then a hash is derived using PBKDF2. The implementation uses

`Rfc2898DeriveBytes` with 100,000 iterations to produce a 256-bit hash of the password. (This is a strong password hashing approach for security.) The salt and hash are converted to Base64 strings for storage.

The new user is then added to the XML: the code creates a `<user>` element, sets its attributes `id` and `role` ("Participant"), and appends child elements for all the collected data. These include `<firstName>`, `<lastName>`, `<email>`, `<university>`, `<passwordHash>`, `<passwordSalt>`, `<createdAt>`, and `<consentAcceptedAt>`. Note that **university** is left blank at creation (the placeholder empty element is added) since the sign-up form does not ask for it. Finally, the XML document is saved (persisting the new user), and the user is redirected to the Login page. The `FormMessage` label might show a success note briefly, but since the code calls `Response.Redirect("~/Account/Login.aspx")` immediately, the typical flow is just to go to the login screen after a successful registration.

Login.aspx (User Sign-In)

The Login page allows existing users to authenticate. Its UI is straightforward: an Email textbox, a Password textbox, and a "Sign in" button (with validation to ensure both fields are filled). In the code-behind (`Login.aspx.cs`), the `BtnLogin_Click` handler performs authentication against the `users.xml` data store.

First, it checks that the users XML file exists – if the application has no users (and thus no `users.xml`), it cannot proceed. In that case it sets an error (for example: "No accounts exist, please create one first.") via the `FormMessage` label and returns. Assuming the file is present, it loads the XML and looks up a user node matching the provided email. This lookup is done in a case-insensitive way (the code likely lower-cases the input email and compares to the stored emails similarly to the registration uniqueness check). If no matching `<user>` is found, or if one is found but the password is incorrect, the login fails – the code sets a generic error message like "Invalid email or password" in `FormMessage` and stops there.

If the email is found, the code retrieves the stored `passwordSalt` and `passwordHash` from the XML. It then hashes the entered password with the same salt (again using PBKDF2 with the same iteration count) and compares the result to the stored hash. If they match, authentication succeeds. The app then creates a session for the user and stores important info in `Session` variables, for example: - `Session["UserId"]` – the user's ID (from the XML `id` attribute). - `Session["Role"]` – the user's role (e.g. "Participant", "UniversityAdmin", or "SuperAdmin"). - `Session["Email"]` – the user's email (used for display and for admin reference).

Finally, the user is **redirected to a role-specific home page**. The code checks the user's role attribute and directs them accordingly: - Participants go to `~/Account/Participant/Home.aspx` - University Admins go to `~/Account/UniversityAdmin/UniversityAdminHome.aspx` - Super Admins go to `~/Account/SuperAdmin/SuperAdminHome.aspx`

This logic ensures each type of user lands in the appropriate section of the site after login. If the role is somehow unrecognized or missing, the code by default also sends the user to the Participant home as a fallback (though in normal operation that shouldn't happen).

Participant Portal

Once a user with the Participant role logs in, they are taken to the participant dashboard section of the site.

Participant Home (Account/Participant/Home.aspx)

This page (accessible only to logged-in Participants) serves as the participant's dashboard. The code-behind (`Home.aspx.cs`) begins by **gating access**: it checks `Session["Role"]` and ensures it equals "Participant". If not (for example, someone tries to access it directly or the session expired), the page immediately redirects to the Login page. This security check is present on all role-specific pages.

On initial load, the participant home checks what event (if any) the participant is associated with. If the user has not yet selected an event to participate in, the interface will prompt them to do so. In practice, the Home page displays the user's current **University** and **Event name** at the top, with a link labeled "Change" next to the event. Clicking "Change" navigates to the event selection page where the user can choose an event (more on that below). If the user has no event yet, this likely appears as "(Select an event)" prompt. The participant's **name** and university are also shown as part of a welcome message (e.g., "Welcome, [Name] from [University]"). The code-behind obtains the user's first name (and possibly last name) from the users XML using the user's ID stored in session, then sets a label or literal on the page. Similarly, it retrieves the university name – which after event selection is stored both in the user profile and in `Session["University"]` – to display the participant's college or institution.

The Participant Home page also provides an overview of the learning content the participant can engage in as part of the selected event. Specifically, it lists the **micro-courses** or modules for that event. The code-behind loads two XML files for this: `microcourses.xml` (which contains definitions of all available courses) and `eventCourses.xml` (which lists which courses are enabled for the participant's chosen event). Using these, it compiles a list of courses relevant to the event. The page likely uses a Repeater control to display each course/module that is part of the event's curriculum, possibly showing the course name and whether it's active. For example, the code iterates through the `<course>` entries in `microcourses.xml` that correspond to the current event's course IDs (found in `eventCourses.xml`). Each course could be displayed as an item (perhaps with a title and description). This gives participants a view of what content ("micro-courses") they should complete for the cyberfair event. (At this stage, completion tracking or detailed interaction isn't described, but the structure is in place to show the list.)

Finally, the participant home likely includes a **Logout** option (e.g., a "Sign out" button). If clicked, it would clear the session (as we see implemented in the admin pages) and redirect back to the login or welcome page. This ensures participants can securely exit the application.

Participant SelectEvent (Account/Participant/SelectEvent.aspx)

If a participant has no event selected (for instance, right after account creation or on first login), they will use the Select Event page to choose which cyberfair event to join. This page is also under Participant-only access (it checks the session role "Participant" in Page_Load).

The SelectEvent page presents a two-step selection form: 1. **Choose University** – A dropdown (`UniversitySelect`) is populated with the list of universities that have events available. On Page_Load,

the code-behind loads `events.xml` and gathers all distinct `<university>` names from the events listed. These are added as options in the University dropdown. The first item of the dropdown is a prompt like "-- Select university --". If the participant's own profile already has a university on file (from a previous selection), the page might auto-select it or skip this step, but typically a new participant will pick their university here. 2. **Choose Event** – Once a university is selected (triggering a post-back `UniversitySelect_SelectedIndexChanged` event), the page populates a second dropdown (`EventSelect`) with the specific events hosted by that university. The code filters `events.xml` for entries where the `<university>` child matches the selected value, and for each event it adds an option (likely using the event's name as text and the event's ID as the value). The user then selects which event (e.g., "2025 CyberFair at XYZ University") they want to join.

After both selections are made, the participant clicks a **Continue** button. The `BtnContinue_Click` handler then executes the following: - It validates that both a university and an event have been chosen (if not, it sets an error like "Pick a course." in the `ScheduleMessage` or similar – note: the code reused a label for messages, the snippet shows a check for courseId selection which is analogous to event selection here). - It saves the selected values into the session: e.g. `Session["University"]` is set to the chosen university name, and `Session["EventID"]` is set to the chosen event's ID. These session variables allow other pages (like the Participant Home) to know what event context the user is in. - It updates the user's record in `users.xml` to store their university if it was previously blank. During sign-up the university was not set, so here the code finds the current user's `<user>` node (using the session `UserId`) in the XML and sets the `<university>` element to the selected name. It then saves the users.xml file. This ensures that on subsequent logins, the user's university is known. (The event ID itself is not persisted in the user profile – the app can infer available events by university.) - Finally, it redirects the participant to the Participant Home (`Home.aspx`) now that an event is chosen.

After this process, the participant's session knows which event to work with, and the Home page will show information for that event (university and courses). The participant can always come back to SelectEvent (via the "Change" link) if they want to switch events, which would repeat the above process.

University Admin Portal

University administrators (users with role "UniversityAdmin") have their own section to manage events and view content.

UniversityAdminHome.aspx (Admin Dashboard)

This is the landing page for a logged-in University Admin. Like other role pages, its Page_Load enforces that the session role is "UniversityAdmin", otherwise it redirects to login. The UniversityAdmin Home serves as a dashboard showing the admin's events and options to manage them.

On loading, the page likely lists the **cyberfair event(s)** associated with that admin's university. Each University Admin is typically responsible for events at their own university. The code-behind (`UniversityAdminHome.aspx.cs`) loads the `events.xml` file and filters events either by the `createdBy` attribute (matching the admin's email) or by the `<university>` element (matching the admin's university name). In the sample data, events have a `createdBy` attribute which is the email of the admin who created them (e.g., `"ua.asu@fia.org"`), and a `<university>` name (e.g., "Arizona State

University"). The Home page likely uses a Repeater or similar control to display a summary of each event (name, date, status, etc.) that this admin can manage.

An important feature is the ability to **create a new event**. If the admin has no events yet, the code appears to automatically create a placeholder “Draft” event for them. In the code-behind, we see logic that creates a new `<event>` element with default attributes: a generated GUID for `id`, `status="Draft"`, `createdAt` timestamp (current UTC), and `createdBy` set to the admin’s email. It also adds child elements like `<university>` (filled with the admin’s university name), `<name>` (possibly a generic name like "New CyberFair"), and `<date>` (which might be blank or a default date). This new event node is appended to `events.xml` and saved. By doing this on first load (or via a “Create Event” button), the system ensures there is at least an event entry that the admin can then edit. In practice, the UI might immediately redirect the admin to the EventManage page to edit the details of this newly created event.

For existing events, the UniversityAdminHome page would show them in a list with perhaps a “Manage” or “Edit” link for each. Clicking that link likely goes to `EventManage.aspx` for that event (often via a query string like `?id=<eventId>` or by storing the ID in session). There might also be a logout button on this page. Indeed, we see a `BtnLogout_Click` handler in the code that clears the session and sends the user back to the login page.

In summary, UniversityAdminHome is the overview page where an admin can see all their events and initiate managing them. It also takes care of event creation (setting up a draft event) so the admin can quickly start configuring it.

EventManage.aspx (Event Management)

This page is the core interface for a University Admin to configure a specific event. It is loaded when an admin wants to edit an event’s details, manage its courses, and schedule sessions. Access is restricted to UniversityAdmin role as well (checked in code-behind).

Event Details: On Page_Load, the code identifies which event is being managed (likely via an `eventId` query parameter or a session variable set before navigating here). It then loads `events.xml` and finds that event’s node by its `id`. The page might display the event’s current properties (name, date, etc.) in form fields for editing, though from the provided markup snippet we mainly saw functionality for courses and sessions. It’s possible the event name or date could be edited here or on the home page; if not, those might remain in the XML as set by default or require a future form.

Course Inclusion (Curriculum): The EventManage page allows the admin to choose which courses (from the library of “microcourses”) are part of this event’s curriculum, and to toggle their availability. It does so by interacting with `eventCourses.xml` and `microcourses.xml`: - The page loads all courses currently associated with this event. In `eventCourses.xml`, each `<event>` node (identified by event `id`) contains multiple `<course>` child entries. Each course entry has an `id` (matching a course in microcourses.xml) and an `enabled="true/false"` flag. The EventManage page uses a Repeater (`CoursesRepeater`) to list these courses. For each course, it likely shows the course name (which it can look up in `microcourses.xml` using the course id) and a checkbox for “Enabled”. In the markup, we see a CheckBox bound to the `enabled` value of each course item. This lets the admin see which courses are active for the event and potentially deactivate some by unchecking them. - The admin can apply changes by

clicking **“Save visibility”** (the button `BtnSaveSwitches`). The code-behind for this button loops through the Repeater items and updates the corresponding `<course>` entries in `eventCourses.xml` to set their `enabled` attribute according to the checkboxes. After updating, it saves the XML file. This way, the curriculum for the event is updated. (For example, if an admin unchecks a course, it might remain listed but marked disabled, or the application might hide it from participants – depending on how the participant side reads the flag.) - The page also allows adding new courses to the event. There is a dropdown `CourseSelect` that contains available courses not yet in the event. On Page_Load, the code populates this with courses from `microcourses.xml` that are not already present under the event’s entry in `eventCourses.xml`. The admin can select one and click **“Add session”** (which as the name suggests, adds a session – see below). It’s a bit subtly named: adding a “session” actually schedules a time for a course, but implicitly, adding a session will also include that course in the event if it wasn’t already. There may also be a separate step to “Add Course” to the event, but given the UI elements, selecting a course and setting a session might do both at once.

Event Sessions (Scheduling): A session represents a scheduled instance of a course within the event (like a workshop or class meeting). The EventManage page provides inputs to schedule sessions: - A **Date/Time picker** (`SessionDateTimeStart`) for the session start time (in local datetime, which the code will convert to UTC). - Text fields for **Room**, **Helper**, and **Capacity**. These correspond to optional details: room/location of the session, a helper or assistant name, and the participant capacity for that session. - The Course dropdown mentioned above to choose which course the session is for.

When the admin clicks **“Add session”** (`BtnAddSession_Click`), the code validates the inputs: it ensures a course was selected and a date/time entered. It then creates a new `<session>` entry in `eventSessions.xml`. Each session node has: - An `eventId` attribute linking it to the event. - A unique `id` (generated GUID) for the session itself. - Child elements for the course ID, start time, end time, room, helper, and capacity. For example, after adding, `eventSessions.xml` might contain:

```
<session eventId="a8c2766fe860463a9d5d600adcbd8c10"
id="47688e66f35047c5b5878fe4ec44cb8c">
  <courseId>c-0002</courseId>
  <start>2025-10-23T06:39:00.0000000Z</start>
  <end>2025-10-26T06:39:00.0000000Z</end>
  <room>MU 201</room>
  <helper>Tracy</helper>
  <capacity>5</capacity>
</session>
```

This example (from the sample data) shows a session for course c-0002 with a start and end date, room, helper, and capacity. In the UI, the admin likely provided the start datetime; the **end datetime** might either be computed (not seen in the form, possibly they assume a default duration or the admin could be expected to update it in XML directly). The code snippet suggests it only explicitly handled start time and left the rest optional. If no end is given via UI, the system might copy the start to end or just omit end (but the XML in the sample had end; possibly they set end equal to start for now or required the admin to manually edit the XML for multi-day events). - The room, helper, capacity fields are taken from the respective TextBoxes if provided. The code likely checks if those text fields are non-empty and, if so, adds the corresponding elements. If a field is left blank, it might skip adding that element or add it empty.

After constructing the `<session>` element, the code appends it under the root `<eventSessions>` and saves the file. It may display a confirmation message on the page like “Session added.” (using something like a `ScheduleMessage` label). **Note:** The current UI does not explicitly list the scheduled sessions back to the admin (we didn’t see a Repeater for sessions), so the admin might not see the session they added on the page itself. Possibly they intended to add a table of existing sessions in the future. As it stands, the data is stored and participants in that event could eventually see the schedule (if a feature was added for that on the participant side).

In summary, the EventManage page lets an admin configure **what courses** are included in an event and **when sessions** for those courses take place. It manipulates three XML files: `events.xml` (for basic event info, though editing name/date might be minimal), `eventCourses.xml` (for course selection toggles), and `eventSessions.xml` (for scheduling sessions). This page is central for a University Admin to set up their cyberfair event’s content and schedule after creation.

Super Admin Portal

The SuperAdmin role is for top-level administrators who oversee the platform. Their interface is simpler, focusing on global settings rather than specific events.

SuperAdminHome.aspx (Super Admin Dashboard)

This is the welcome page for a logged-in Super Admin. Like others, it ensures `Session["Role"]` is “SuperAdmin” on load. The SuperAdmin home likely just shows a greeting and provides navigation to management functions. In this case, the primary function provided is managing **Certification Rules**. The ASPX markup contains a link: “Open Certification Rules” which navigates to `CertificationRules.aspx`. The page might also display the super admin’s name or email; indeed, we see a Literal control `WelcomeName` used to greet the user. The code-behind sets `WelcomeName.Text` to the super admin’s email from session (since the Super Admin’s personal name might not be stored, using email is a fallback). There isn’t much else on this page besides possibly a general message and the link(s) to admin tools. It serves as a straightforward starting point for super admin tasks.

CertificationRules.aspx (Manage Certification Rules)

The Certification Rules page allows a Super Admin to define and modify rules for certification. “Certification rules” in context likely means criteria or descriptions of certifications that participants can earn (for example, a rule might outline what a participant needs to do to get a “Phishing Awareness” certificate). These rules are stored in the `App_Data/certificationRules.xml` file.

When the page loads, it reads all `<rule>` entries from `certificationRules.xml` and displays them in a list. The ASPX uses a Repeater `RulesRepeater` to list existing rules. Each rule in the XML has attributes `id` and `version`, and child elements like `<name>` and `<description>`. The Repeater likely shows the rule ID, name, and version, and might provide “Edit” and “Delete” command buttons for each item (the `OnItemCommand` is set on the Repeater for handling these actions in code).

Below the list, there is a form for adding or editing a rule. The form fields include: - **RuleId** (TextBox for the rule's identifier) - **RuleName** (TextBox for the rule's name/title) - **RuleDescription** (TextBox, possibly multiline, for the description) - **RuleVersion** (TextBox for version number or code)

There is a **Save** button (`BtnSave`). The code-behind handles several scenarios: - **Ensure XML exists:** Similar to other modules, it has an `EnsureXml()` that will create an initial `certRules` root element if `certificationRules.xml` doesn't exist. This writes something like `<?xml version='1.0'?><certRules version='1'></certRules>` on first run. - **Listing rules:** On Page_Load (when not postback), it loads the XML and binds the `RulesRepeater` to all `<rule>` entries. Each rule's data is accessible to the item template (e.g., using `<%# XPath("name") %>` for name, etc.). - **Edit command:** If the user clicks "Edit" on a particular rule in the list, the `RulesRepeater_ItemCommand` will catch an `"Edit"` command. The code then finds that rule in the XML (using an XPath by id) and populates the form fields with the rule's details. For instance, it sets `RuleId.Text` to the rule's id attribute, `RuleName.Text` to the `<name>` content, `RuleDescription.Text` to the `<description>`, and `RuleVersion.Text` to the version attribute. This allows the admin to modify the fields and then click Save. - **Delete command:** If the user clicks "Delete" on a rule, the ItemCommand handler will locate the rule node in the XML and remove it (e.g., `parentNode.RemoveChild(ruleNode)`). Then it saves the XML file and rebinds the list (so it disappears from the UI). - **Save (add/update) command:** When the admin fills in the form and clicks **Save**, the `BtnSave_Click` handler runs. It loads the XML (and calls EnsureXml just in case). It then requires that a Rule ID is provided (the ID serves as the unique key for rules). The code checks the XML for an existing rule with that id. - If a rule with that ID already exists, it means the admin is updating an existing rule. The code will update that node's fields: set its `version` attribute to the value from the form (or increment it), update the `<name>` text and `<description>` text from the form inputs. - If no such rule exists, this is a request to add a new rule. The code creates a new `<rule>` element, sets the `id` and `version` attributes from the form, and creates `<name>` and `<description>` sub-elements with the provided text. It then appends this new rule under the root `<certRules>` element. After either adding or updating, it saves the `certificationRules.xml` file. The form may be cleared and the list is refreshed (so the new/updated rule appears in the Repeater).

For example, the sample `certificationRules.xml` provided shows a rule:

```
<rule id="2" version="1">
  <name>Phishing Awareness</name>
  <description>Rules needed for Phishing Awareness Certification for
participants...</description>
</rule>
```

Using the interface, a super admin could change its description or name, bump the version to "2", or add a new rule with a new id for another topic. The page ensures all this is done through a simple UI instead of manually editing the XML.

This page encapsulates a CRUD interface for the certification rules. It's a good example of using ASP.NET Web Forms data binding and event commands to manage an XML-backed list.

XML Data Files (App_Data)

This project uses XML files (in the `App_Data` folder) as a lightweight database. Each main aspect of the system has a corresponding XML: - **users.xml:** Stores all user accounts. Each `<user>` has attributes `id` (unique user ID) and `role` (Participant, UniversityAdmin, or SuperAdmin). Inside each user element are fields: `<firstName>`, `<lastName>`, `<email>`, `<university>`, `<passwordHash>`, `<passwordSalt>`, `<createdAt>` (account creation timestamp), and `<consentAcceptedAt>` (when the user agreed to terms). This file is used by the CreateAccount and Login processes to add and verify users. For example, after a sign-up, a user entry might look like:

```
<user id="d4f8c9...c3a" role="Participant">
  <firstName>Jane</firstName>
  <lastName>Doe</lastName>
  <email>jdoe@university.edu</email>
  <university>University of X</university>
  <passwordHash>Base64HashHere==</passwordHash>
  <passwordSalt>Base64SaltHere==</passwordSalt>
  <createdAt>2025-10-10T00:00:00Z</createdAt>
  <consentAcceptedAt>2025-10-10T00:00:00Z</consentAcceptedAt>
</user>
```

The application reads from and writes to this file when registering users or logging them in. - **events.xml:** Contains the list of events (cyberfair events) configured in the system. Each `<event>` has a unique `id` (GUID string), a `status` (e.g., Draft or Published), `createdAt` timestamp, and `createdBy` (the admin's email who created it). Child elements hold the event's details, such as `<university>` (the hosting university name), `<name>` (event title), and `<date>` (scheduled date or date range of the event). University Admins create and manage events via this file. Initially, events may be created as "Draft" with placeholder data for the admin to edit. Participants use this file (via the SelectEvent page) to choose which event to join, typically filtered by their university. - **microcourses.xml:** Acts as a library of course content or modules (referred to as "micro-courses"). Each `<course>` in this file has an `id` (could be a code like "c-0001" or a GUID), a `status` (Draft, Published, etc.), a `createdAt`, and possibly other attributes. It likely includes child elements for the course's title, content, or description (though the snippet we saw only showed the structure, not full content). Super Admins would define these courses (e.g., general cybersecurity topics) which University Admins can then include in their events. In the current UI, there isn't a dedicated page to edit microcourses (that might be a future feature), so the file might be pre-populated or maintained manually. UniversityAdmin's EventManage page reads from this file to list available courses and to get course names for display. - **eventCourses.xml:** Maps events to the courses offered in them. The structure is an `<eventCourses>` root, containing multiple `<event id="...">` elements. Inside each event element are multiple `<course>` entries, each with an `id` (matching a course in microcourses.xml) and an `enabled` flag. For example:

```
<event id="a8c2766fe860463a9d5d600adcbd8c10">
  <course id="c-0002" enabled="true"/>
```

```
  <course id="5de6f8cad108480a847e89f82df11049" enabled="true"/>
</event>
```

This means that for the event with that GUID, two courses are part of it (and currently enabled). The EventManage page uses this file to populate the course list and to save any enable/disable changes. If an admin adds a new course to an event, the application would insert a new `<course>` node here under the corresponding event (with `enabled="true"` by default). - **eventSessions.xml:** Stores scheduled sessions (time-specific offerings) for courses within events. The root is `<eventSessions>` containing `<session>` entries. Each `<session>` has an `eventId` attribute linking it to an event, an `id` as a unique session identifier, and child elements for details: `<courseId>` (which course this session is for), `<start>` (start datetime in ISO format UTC), `<end>` (end datetime in ISO, if applicable), `<room>` (location), `<helper>` (assistant or facilitator), and `<capacity>` (max participants). The University Admin schedules sessions via the EventManage page, which appends sessions to this file. Participants could later view this to see when and where sessions are happening (though the UI for participants to see session schedules isn't described, it could be added using this data). - **certificationRules.xml:** Contains the rules for certifications available in the program. The root `<certRules>` holds multiple `<rule>` entries. Each rule has an `id` (likely a short number or code), a `version` (for tracking changes to the rule set), and typically child elements `<name>` and `<description>` (and possibly others like criteria, but only name/description are shown). Super Admins manage this file through the CertificationRules page. This allows updating descriptions or adding new certification rule entries without touching the file manually. Participants or admins elsewhere in the application might use these rules to determine when a participant qualifies for a certificate, but such logic isn't detailed in the code we reviewed – it appears the scope for now is just to maintain the content of the rules.

All these XML files are simple data stores that the application reads at runtime and writes back to as users perform actions. They are all versioned (most have a `version="1"` in the root element or as an attribute), which could be used for migration if the data schema changes in the future. The use of `Server.MapPath("~/App_Data/..")` in code gets the physical file path to these XML files on the server, and standard .NET `XmlDocument` and `XmlElement` APIs are used for manipulation. Because the data is stored in App_Data, it is protected from direct HTTP access (as per ASP.NET conventions), ensuring that users cannot download these files via URL – they can only be accessed through the application's code.

Conclusion

Overall, the project is structured as an ASP.NET Web Forms application with a clear separation of concerns by role: - Public pages (Welcome, Login, Registration) handle user entry and account creation. - Participant pages allow event selection and show event content (courses). - University Admin pages allow creating events, selecting course content for events, and scheduling sessions. - Super Admin pages allow managing high-level configuration like certification rules (and presumably could be extended to manage the course library or other global data).

Important concepts illustrated in the code include: - **Session-based authentication and authorization:** The app uses Session to keep track of the logged-in user and their role, and each page's code-behind checks this to restrict access. - **XML as a database:** Instead of a SQL database, XML files store the data. The code demonstrates reading/writing XML for each feature (with helper functions like `EnsureXml` to initialize files and using XPath for searches like finding users by email). - **Security:** Passwords are never stored in plain

text – the use of salted PBKDF2 hashing provides secure password storage. Also, sensitive operations double-check conditions server-side (e.g., consent checkbox, input validation) even though ASP.NET validators run on the client, which is good practice. - **Separation of Roles:** The folder structure (`Account/Participant`, `Account/UniversityAdmin`, `Account/SuperAdmin`) and code logic keep each user role's functionality distinct. This makes the flow easier to follow and maintain per role. - **ASP.NET Web Forms features:** The pages use standard Web Forms controls (TextBox, Button, DropDownList, Repeater, Validators, etc.) and code-behind events to implement the interactive logic. The use of `OnItemCommand` in repeaters, `OnSelectedIndexChanged` for dropdowns, and server-side redirects with `Response.Redirect` are all typical patterns in Web Forms development.

By walking through each page and its associated logic, a new developer can understand how a user moves through the system (from signing up or logging in, to performing actions according to their role) and how the data flows to and from the XML storage. This understanding will help in maintaining the code or extending it – for example, adding new fields to the registration (and users.xml), or creating a page for Super Admin to manage the microcourses library, would follow the same patterns seen here. The code is organized in a way that each piece is responsible for a specific part of the application's functionality, making it easier to jump in and make targeted changes in the future.
