TRADECRAFT TECHNOLOGIES, INC

AMPS Developers Guide

Introduction

Development for the AMPS system consists of a wide variety of technologies and technology platforms (see AMPS Architecture Diagram for reference). There are several integration points that tie them all together to ensure the data is used in a logical manner for providing a rich user experience (for more details regarding the integrations, please see the Integrator's Guide for more details). Given the size and complexity of the system, it is imperative that a set of standards be defined to ensure a certain level of commonality and continuity throughout the source code.

Development/Coding Standards

General (all technologies):

- Variable names should be English based and should be a clear description of what the variable is or is to be used for in the application
- Program logic should be written in an efficient manner to ensure the application executes as quickly as possible
- Libraries should be created in a logic manner for the programming language that is being used. For example, the Web API should have Models and Controllers, the React front end should have Actions, Containers, Components, etc.
- All source code should be first tested by the developer in their own local development environment before being checked into the source code repository or placed on the shared development server
- Thorough testing should take place on the development server prior to moving it to the end user's testing environment
- Code should be commented with meaningful comments throughout the application to allow other developers quickly determine the purpose and flow of your code
- All connections to the other components (i.e. API, database, etc.) need to be maintained in a configuration file that can be updated on the fly without requiring a recompile of any of the applications components

C# Web API:

- GET, PUT, DELETE, POST actions should be idempotent
- POST actions that create new resources should not have unrelated side-effects
- Follow the HTTP specification when sending a response
- Capture exceptions and return a meaningful response to clients
- Handle exceptions consistently and log information about errors
- Distinguish between client-side errors (HTTP 4xx) and server-side errors (HTTP 5xx)
- Ensure that each request is stateless

- When adding new functionality to the API, add the appropriate XML documentation – these automatically populate in the Swagger front end
- When adding a new parameter to an existing endpoint, if possible, do so in a manner that won't break any existing calls to the function (i.e. allow it to be null or assign a default value)
- When modifying a response object, if possible, minimize impact to any existing calls of the service (add properties versus reusing/repurposing them). Only remove a property if you are certain that all the other calls can handle the change.
- Validate all input to ensure the data is clean from any web attacks and that the data is acceptable for it's intended use (i.e. make sure there aren't alphabetic characters in a numeric variable, etc.)
- Carefully consider the visibility of your methods and data members – only publicly expose the ones that should be made available to other parts of the program.
- Implement IDisposable if a class uses unmanaged/expensive resources, owns disposable objects or subscribes to other objects
- All switch statements shall have a default label as the last case label
- Use the correct way of casting
- Use standard exceptions
- Consider using Any() to determine whether an IEnumerable is empty
- Do not use ToUpper() or ToLower() for case insensitive string comparison
- Use List<> instead of ArrayList especially for value types
- Do not use letters that can be mistaken for digits, and vice versa
- Use logging throughout all of the functions
- Use Entity Framework 6 for all interactions with the database. The only exception are stored procedures which tend to make more sense for longer running queries or queued items
- OAuth2 should be used for authentication using bearer tokens for security
- All endpoints should be carefully considered whether they should be limited to specific user role(s) – for example: limited to only admin user roles
- If an unauthenticated user attempts to access an endpoint, the return should be HTTP 401
- Use standard HTTP response codes when interacting with users; common HTTP response codes:
  - HTTP 400 – Bad Request – The request was invalid or validation failed.
  - HTTP 401 – Unauthorized – The request requires a user authentication.
  - HTTP 500 – Internal Server Error - Some exception or error occurred in your action. So use 500 for failures.
  - HTTP 200 – OK – Everything is good.
  - HTTP 201 – OK – New resources has been created
  - HTTP 202 – Accepted – Request is accepted and it's good for processing.

React.js

- Always use prop-types to define all the props in your components
- Validate all user input to ensure it meets the intended needs of any given field, also check with the database developers to see if there are any length or data restrictions that can be implemented in the front end
- Non-static data should be retrieved and/or saved via the API – direct database connections should not be used

- Visual design and layouts should be consistent throughout the application, color schemes need to match the overall theme
- Keep the layout/structure of the code consistent (components are all under a folder with logical breakdown of the individual files, actions have their own folder, etc.)
- CSS styles should be in separate css or scss files – not inline
- Ensure proper error handling is in place; proper user data entry validations as well as system errors
- Always put spaces around operators (= + - * /), and after commas
- Always use 4 spaces for indentation of code blocks
- Always end a simple statement with a semicolon
- For complex/compound statements (functions, loops, conditionals):
    - Put the opening bracket at the end of the first line
    - Use one space before the opening bracket
    - Put the closing bracket on a new line, without leading spaces
    - Do not end a complex statement with a semicolon
    - Exception: empty blocks can be concise (on the same line)
- For objects (JSON or otherwise):
    - Place the opening bracket on the same line as the object name.
    - Use colon plus one space between each property and its value.
    - Use quotes around string values, not around numeric values.
    - Do not add a comma after the last property-value pair.
    - Place the closing bracket on a new line, without leading spaces.
    - Always end an object definition with a semicolon
- Source files are encoded in UTF-8
- One statement per line of code
- Declare all local variables with either const or let. Use const by default, unless a variable needs to be reassigned. The var keyword must not be used.
- Use trailing commas versus leading commas
- Do not mix quoted and unquoted keys in an object or JSON
- Within a switch block, each statement group either terminates abruptly (with a break, return or thrown exception), or is marked with a comment to indicate that execution will or might continue into the next statement group. Any comment that communicates the idea of fall-through is sufficient (typically // fall through). This special comment is not required in the last statement group of the switch block.
- Exceptions are an important part of the language and should be used whenever exceptional cases occur. Always throw Errors or subclasses of Error: never throw string literals or other objects. Always use new when constructing an Error.
- Do not use the eval operator