Jyros – Not Atlassian

Overview

This project is a web-based application designed as a clone of Jira, with an AI-powered feature to estimate the difficulty of each ticket.

Tech Stack

- **Frontend:** React with ShadCN/UI component library
- **Backend:** ASP.NET with SQL Server, utilizing Entity Framework
- **AI:** Pytorch

Frontend

The frontend is built using React and is structured into four main pages:

- **Log In Page**
- **Backlog**
- **Board**
- **Dev Availability**

Each page is a separate React component and utilizes various subcomponents, including a sidebar navigation panel. The UI components are managed with ShadCN/UI for a consistent and modern user interface.

App Component

The App component is the main entry point of the application, responsible for setting up routing and managing the authentication context. It uses React Router for navigation and includes private routes for authenticated users.

Component Structure

- **AppProvider**: Wraps the entire app, providing the authentication context to all child components.
- **BrowserRouter**: The routing component that enables navigation within the app.
- **Routes and Route**: Define the routes and the corresponding components for different paths in the app.

Routes

- **/**: Redirects to /board, serving as the default route.
- **/login**: Renders the LoginPage component, allowing users to log in.
- **/board**: Protected by PrivateRoute. Renders the Board component for authenticated users.
- **/backlog**: Protected by PrivateRoute. Renders the BacklogPage component for authenticated users.
- **/availability**: Protected by PrivateRoute. Renders the Availability component for authenticated users.

PrivateRoute:

The PrivateRoute component wraps protected routes to ensure only authenticated users can access them. If a user is not authenticated, they are redirected to the login page.

LoginPage Component

This component renders a login and signup page, which allows users to toggle between login and signup modes. The page is part of the user authentication flow.

State Variables

- **isLogin** (boolean): Tracks whether the user is in login or signup mode. Defaults to true (login).
- **error** (string | null): Stores any error message, initialized as null.
- **navigate**: The useNavigate hook is used to programmatically navigate after a successful login.

Functions

- **handleSubmit(username, password)**: This function is triggered when the form is submitted. It attempts to log the user in using the LogIn function from the context. Upon successful login, it redirects the user to the /backlog page. Errors are caught and logged.
- **toggleAuthMode()**: Toggles the isLogin state, switching between login and signup modes.

UI Structure

- A centered container (div) displays either a "Log In" or "Sign Up" heading based on isLogin.
- The SignUpForm component is rendered with the following props:
    - isLogin: Determines the current mode (login/signup).
    - onSubmit: Passes the handleSubmit function for form submission.
    - onToggle: Passes the toggleAuthMode function to switch between login/signup views.

SignUpForm Component

The SignUpForm component handles both the login and signup forms, dynamically adjusting based on the isLogin prop. It accepts the necessary props to handle user submission, password confirmation (for signup), and toggling between login and signup modes.

Props

- **isLogin** (boolean): Determines whether the form is in login or signup mode.
- **onSubmit** ((username: string, password: string, confirmPassword?: string) => void): A function that handles form submission. For login, it requires username and password; for signup, it additionally requires confirmPassword.
- **onToggle** (() => void): A function to toggle between login and signup modes.

State Variables

- **username** (string): Stores the entered username.
- **password** (string): Stores the entered password.
- **confirmPassword** (string): Stores the confirmed password (used only in signup mode).

Context

- **SignIn**: Accessed from the AppContext to handle user signups.

Navigation

- **navigate**: The useNavigate hook from React Router is used to navigate the user after successful signup (redirecting to /login).

Functions

- **handleSubmit**: This function handles the form submission:
    - If isLogin is true, it calls onSubmit with the username and password.
    - If isLogin is false, it checks if the password and confirmPassword match before calling SignIn from the context for signup. If successful, it navigates the user to the login page.
    - Prevents the default form submission behavior to avoid page reload.

UI Structure

- **Form Fields**: The form contains the following fields:
    - **Username**: An input for the user's username.
    - **Password**: An input for the password.
    - **Confirm Password** (only in signup mode): A field to re-enter the password for confirmation.
- **Submit Button**: A button that displays either "Log In" or "Sign Up" depending on the mode.
- **Toggle Button**: A button at the bottom to toggle between login and signup modes, triggering the onToggle function.

FilterableTaskTable Component Documentation

The FilterableTaskTable component is a key part of a task management system, where users can manage tasks, filter by status, search by title, and group tasks by shifts. It offers functionality for filtering, sorting, and selecting tasks, as well as creating new shifts and tasks.

Key Features:

- **Task Filtering**: Users can filter tasks based on status (e.g., "To Do", "Cooking", "In Plating", "Bon appétit").
- **Task Searching**: Allows searching by task title with case-insensitive support.
- **Task Sorting**: Sorting tasks by title, priority, and status with ascending or descending options.
- **Shift Grouping**: Tasks are grouped by shift, showing task counts by status for each shift.
- **Task Selection**: Users can select individual tasks or select all tasks at once.
- **Create Shift**: Users can create new shifts with a name, start and end dates, and a description.
- **Create Task**: New tasks can be created with a title, shift, priority, and status.

State Management:

- **tasksLocal**: Stores the list of tasks fetched from the server.
- **shiftsLocal**: Stores the list of shifts fetched from the server.
- **searchQuery**: The search query string for filtering tasks by title.
- **selectedFilters**: Tracks which filters are active for the task status.
- **selectedTasks**: A set of selected task IDs, used for bulk actions.
- **sortConfig**: Manages the current sorting criteria for tasks (e.g., by title, priority, or status).
- **isModalOpen**: Controls the visibility of the shift creation modal.
- **isTaskModalOpen**: Controls the visibility of the task creation modal.

Lifecycle Methods:

- **useEffect** hooks are used to fetch the tasks and shifts when the component mounts or when the relevant context data changes.

Key Functions:

- **handleFilterChange**: Updates the selected filters when a checkbox is clicked in the filter dropdown.
- **handleSort**: Updates the sorting configuration based on the column clicked.
- **handleSelectAll**: Selects or deselects all tasks.
- **handleTaskSelect**: Selects or deselects an individual task.
- **handleCreateShift**: Creates a new shift and updates the global state via context.
- **handleCreateTask**: Creates a new task and updates the global state via context.

UI Structure:

- **Search Bar**: Located at the top, this allows users to filter tasks by title.
- **Filter Dropdown**: Allows users to filter tasks by different statuses (e.g., "To Do", "Cooking").
- **Task Table**: Displays tasks grouped by shift. Each task row includes checkboxes, titles, priorities, and statuses.
- **Task Modal**: A form where users can create a new task, opened when the "Create Task" button is clicked.
- **Shift Modal**: A form for creating a new shift, opened when the "Create Shift" button is clicked.

Task Row Display:

- **Task ID**: Displays the task's ID.
- **Title**: The title of the task.
- **Priority**: Represented by an icon (ForkIcon, Utensils, or UtensilsCrossed).

- **Status**: Displays the current status (e.g., "To Do", "Cooking") with a colored badge.

Board Component

The Board component represents a task management board where users can drag and drop tasks between different columns (e.g., "To Do", "Cooking", "In Plating", "Bonne Appetit"). It fetches tasks, displays them in columns, and updates their state when moved.

State Variables

- **columns (ColumnState)**: Manages the tasks within each column (To Do, Cooking, In Plating, Bonne Appetit).
- **taskId (number)**: Stores the ID of the currently selected task for viewing.
- **viewOpen (boolean)**: Tracks whether the task detail view is open.
- **storyTickets (any)**: Stores the list of fetched tickets.

Context

- **AppContext**: Provides access to tickets, fetchTickets, and updateTicketStatus functions.

Functions

- **useEffect()**:
    - Fetches tickets when the component mounts and whenever the tickets state changes.
    - Updates the task list based on the fetched tickets and organizes them into columns.
- **onDragEnd(result)**: Handles drag-and-drop logic:
    - Updates the state of a task when it's moved between columns.
    - Calls updateTicketStatus to reflect the change in the backend.

UI Structure

- **Sidebar**: Displays a sidebar for navigation.
- **Header**: Contains the project name and a button for additional actions.
- **Columns**: Each column (To Do, Cooking, In Plating, Bonne Appetit) displays a list of tasks. Tasks can be clicked to open a detailed view.
- **Task Cards**: Each task card displays the task content, priority icon, and an avatar of the assignee.

- **Ticket Create**: A button to create new tasks.
- **Ticket View**: Displays detailed information about a selected task when clicked.

ShiftAvailability Component

The ShiftAvailability component manages developers' shift availability and adjustments. It allows users to update availability points for each developer, add or remove adjustments, and select a shift to view the associated data.

State Variables

- **currentShift (number)**: Tracks the currently selected shift.
- **developers (Developer[])**: Stores the list of developers and their availability points.
- **totalDays (number)**: Tracks the total number of availability points across all developers.
- **newAdjustment (Adjustment)**: Stores the current adjustment to be added.
- **adjustments (Adjustment[])**: Stores the list of adjustments added by the user.

Context

- **AppContext**: Provides access to various functions, including fetching shift data (fetchUsersInShift, fetchShifts, fetchUserAvailability), managing adjustments (addAdjustment, setAdjustments), and updating developer availability (updateAvailability).

Functions

- **useEffect**:
    - Fetches shift, user, and availability data when the component mounts or when the shift changes.
    - Organizes the developers' data based on the availability points.
- **handleUpdateAvailability**: Updates the availability points for each developer and calls updateAvailability to persist the changes.
- **handleAddAdjustment**: Adds a new adjustment if valid, updates the total days, and calls addAdjustment to persist the change.
- **handleRemoveAdjustment**: Removes an adjustment from the list and updates the total days accordingly.
- **onShiftChange**: Changes the current shift and resets the state variables accordingly.

UI Structure

- **Shift Selector**: A dropdown to select the current shift from available shifts.
- **Developer Availability**: Displays input fields for developers to modify their availability points.
- **Total Availability and Adjustments**: Shows the total availability points, adjustment points, and the remaining available days after adjustments.
- **Adjustment Form**: A form to add adjustments by specifying the number of days and the reason.
- **Adjustment List**: Displays a list of added adjustments with options to remove them.

Ticket Creation Form Component Documentation

This component enables the creation of tickets with attributes like title, description, priority, and status. It uses a dialog box that includes input fields and dropdowns for these attributes, along with features for assigning users, choosing priorities, and submitting tickets.

State Variables:

- **open**: Controls dialog visibility.
- **title, description, assignee, team, etc.**: Ticket attributes.
- **priority, status**: The ticket's priority and status.
- **showAnimation**: Controls the display of an animation after ticket creation.
- **shifts**: List of shifts fetched on mount.

Functions:

- **handlePriorityChange**, **handleStatusChange**: Updates the ticket's priority and status.
- **getStatusColor**: Returns the status color.
- **getPriorityIcon**: Returns an icon based on priority.
- **sendTicketCreationRequest**: Sends ticket data to the backend.
- **handleTicketCreation**: Submits the ticket and triggers animation.
- **handleCancel**: Closes the dialog without submitting.

Components:

- **Dialog**: Modal container for the form.
- **Input, Textarea, Select**: Form fields for ticket details.

- **Button**: Triggers actions like submit or cancel.
- **CrossingKnives**: Animation after ticket creation.
- **EstimationPopup**: Estimation dialog for story points.

EstimationPopup Component

The EstimationPopup component estimates story points for a ticket based on its title and description. It uses an API to fetch the estimation and displays a message to the user, showing either the estimated value or a warning message if the title or description is missing.

Props:

- **title** (string): The title of the ticket.
- **description** (string): The description of the ticket.
- **handleExit** (function): Function to exit the popup.
- **setStoryPlates** (function): Function to set the estimated story points for the ticket.

State Variables:

- **estimation** (number): Holds the estimated story points.
- **displayedText** (string): Message to display in the popup.
- **isTyping** (boolean): Flag to show the typing indicator while fetching estimation.

Functions:

- **handleAccept**: When clicked, this will set the estimated story points if the estimation is valid and close the popup.
- **useEffect**: Fetches the estimation from an API when the title and description change and updates the message accordingly.

Effects:

- On mount or when the title or description change, it fetches the estimated story points via a fetch request to an API.
- Displays a typing indicator while fetching the data.

Styling:

- The component applies custom CSS from EstimationPopup.css to style the bot icon, message container, and buttons.

Returns:

Renders a popup with:

- A bot icon.
- A message that is either the estimation or a warning message.
- A button to accept or cancel the estimation

Setup:

Firstly, ensure you have nodeJS installed and a text editor, like VSCode. After that, open the project in terminal, navigate inside the Jyros directory and run the following command:
npm install
To run the application use the following command:
npm run dev

Backend

The backend is developed using ASP.NET and follows a structured architecture:

- **Controllers:** Expose REST API endpoints
- **Repositories:** Handle database operations
- **Entity Framework:** Manages database interactions with SQL Server

Entities

The application works with the following entities:

- **Adjustment**
- **Sprint**
- **Story**
- **Team**
- **User (TBD)**
- **TeamMemberAvailability**

Each frontend page has a corresponding controller, and each entity has a repository. A controller may interact with multiple repositories, and a repository can be used by multiple controllers.

Adjustment Class

Properties:

- **Id** (int): A unique identifier for the adjustment record.
- **SprintId** (int): The ID of the sprint associated with the adjustment.
- **AdjustmentPoints** (int): The number of points adjusted for the sprint.
- **Reason** (string): The reason for the adjustment. This is a required field and cannot be null.

Navigation Properties:

- **Sprint** (Sprint?): A reference to the associated Sprint object. This property is marked with the JsonIgnore attribute to prevent it from being serialized into JSON.

Sprint Class

Properties:

- **SprintId** (int): A unique identifier for the sprint.
- **Name** (string): The name of the sprint. This is a required field and cannot be null.
- **Goal** (string?): A description of the goal for the sprint. This is optional.
- **StartDate** (DateOnly): The start date of the sprint.
- **EndDate** (DateOnly): The end date of the sprint.
- **Status** (string?): The current status of the sprint (e.g., "Active", "Completed"). This is optional.
- **TeamId** (int?): The ID of the team associated with the sprint. This is optional.

Navigation Properties:

- **Stories** (ICollection<Story>): A collection of Story objects associated with the sprint. This is excluded from JSON serialization using the JsonIgnore attribute.
- **TeamMemberAvailabilities** (ICollection<TeamMemberAvailability>): A collection of TeamMemberAvailability objects representing the availability of team members during the sprint. Also excluded from JSON serialization.

- **Adjustments** (ICollection<Adjustment>): A collection of Adjustment objects associated with the sprint, such as points adjustments. This property is also excluded from JSON serialization.
- **Team** (Team?): A reference to the associated Team object, representing the team assigned to the sprint. Excluded from JSON serialization.

Default Values:

- **Stories, TeamMemberAvailabilities, Adjustments**: Initialized as empty lists to ensure they are never null, avoiding null reference errors.

Story Class

Properties:

- **StoryId** (int): A unique identifier for the story.
- **Title** (string): The title of the story. This is a required field and cannot be null.
- **Description** (string?): A description of the story. This field is optional.
- **Status** (string?): The current status of the story (e.g., "To Do", "In Progress", "Completed"). This field is optional.
- **Priority** (int): The priority level of the story. A higher number typically indicates a higher priority.
- **ParentId** (int?): The ID of the parent story, if this is a sub-story or task under a larger story. This field is optional.
- **SprintId** (int?): The ID of the sprint that this story is associated with. This field is optional.
- **CreatedBy** (int?): The ID of the user who created the story. This field is optional.
- **StoryPoints** (int?): The number of story points associated with the story. This field is optional and may be used for estimation purposes.

Navigation Properties:

- **CreatedByNavigation** (User?): A reference to the User who created the story. This is excluded from JSON serialization using the JsonIgnore attribute.
- **InverseParent** (ICollection<Story>): A collection of child stories (sub-tasks) for this story. These are stories that have this story as their parent. This is excluded from JSON serialization.
- **Parent** (Story?): A reference to the parent story, if this story is a sub-task or dependent on another story. This is excluded from JSON serialization.
- **Sprint** (Sprint?): A reference to the associated Sprint object, indicating which sprint this story is part of. This is excluded from JSON serialization.
- **Users** (ICollection<User>): A collection of User objects representing the users assigned to this story. This is excluded from JSON serialization.

Default Values:

- **InverseParent, Users**: Initialized as empty lists to ensure they are never null, avoiding null reference errors.

Team Class

Properties:

- **TeamId** (int): A unique identifier for the team. This field is excluded from JSON serialization with the JsonIgnore attribute.
- **TeamName** (string): The name of the team. This is a required field and cannot be null.
- **TeamDescription** (string?): A description of the team. This field is optional.
- **TeamLeadId** (int?): The ID of the user who leads the team. This field is optional.

Navigation Properties:

- **Sprints** (ICollection<Sprint>): A collection of Sprint objects associated with this team. This is excluded from JSON serialization to avoid circular references and is initialized as an empty list.
- **TeamLead** (User?): A reference to the User object representing the team lead. This is excluded from JSON serialization.
- **Users** (ICollection<User>): A collection of User objects representing the members of the team. This is excluded from JSON serialization and is initialized as an empty list.

Default Values:

- **Sprints, Users**: Initialized as empty lists to ensure they are never null, thus avoiding null reference errors.

TeamMemberAvailability Class

Properties:

- **Id** (int): A unique identifier for the team member availability record. This field is excluded from JSON serialization with the JsonIgnore attribute.
- **UserId** (int): The ID of the user (team member) associated with the availability record.

- **SprintId** (int): The ID of the sprint that the availability record is associated with.
- **AvailabilityPoints** (int): The number of availability points for the team member during the given sprint. This represents the available capacity or effort the team member can contribute to the sprint.

Navigation Properties:

- **Sprint** (Sprint?): A reference to the Sprint object associated with this availability record. This is excluded from JSON serialization with the JsonIgnore attribute.

Annotations:

- **JsonIgnore**: The Id and Sprint properties are excluded from JSON serialization to avoid circular references and unnecessary data being included in API responses.

User Class

Properties:

- **UserId** (int): A unique identifier for the user.
- **Username** (string): The username of the user.
- **Password** (string): The password for the user account.

Navigation Properties:

- **Stories** (ICollection<Story>): A collection of Story objects that are associated with this user. This property is excluded from JSON serialization with the JsonIgnore attribute.
- **Teams** (ICollection<Team>): A collection of Team objects that the user is a member of. This property is excluded from JSON serialization with the JsonIgnore attribute.
- **StoriesNavigation** (ICollection<Story>): Another collection of Story objects, possibly with different relationships or context. Excluded from JSON serialization with the JsonIgnore attribute.
- **TeamsNavigation** (ICollection<Team>): Another collection of Team objects that the user is a member of. Excluded from JSON serialization with the JsonIgnore attribute.

Default Values:

- The Stories, Teams, StoriesNavigation, and TeamsNavigation collections are initialized as empty lists by default.

IRepository<T> Interface

Generic Type Parameter:

- **T**: Represents the type of entity managed by the repository. It must be a class (where T : class).

Methods:

1. **Task<IEnumerable<T>> GetAll()**:
    - **Description**: Retrieves all entities of type T from the data store.
    - **Returns**: A task that, when completed, returns an IEnumerable<T> collection containing all entities.
2. **Task<T> GetById(int id)**:
    - **Description**: Retrieves a single entity of type T by its unique identifier (id).
    - **Parameters**:
        - id (int): The unique identifier of the entity.
    - **Returns**: A task that, when completed, returns the entity of type T with the specified ID.
3. **Task<T> Add(T entity)**:
    - **Description**: Adds a new entity of type T to the data store.
    - **Parameters**:
        - entity (T): The entity to be added.
    - **Returns**: A task that, when completed, returns the added entity, including any generated properties (e.g., ID).
4. **Task<T> Update(T entity)**:
    - **Description**: Updates an existing entity of type T in the data store.
    - **Parameters**:
        - entity (T): The entity to be updated.
    - **Returns**: A task that, when completed, returns the updated entity.
5. **Task<T> Delete(int id)**:
    - **Description**: Deletes an entity of type T by its unique identifier (id).
    - **Parameters**:
        - id (int): The unique identifier of the entity to be deleted.
    - **Returns**: A task that, when completed, returns the deleted entity.
6. **Task<IEnumerable<T>> GetPaginated(int page, int pageSize)**:
    - **Description**: Retrieves a paginated list of entities of type T from the data store.

    - **Parameters**:
        - page (int): The page number to retrieve.
        - pageSize (int): The number of entities per page.
    - **Returns**: A task that, when completed, returns an IEnumerable<T> containing a subset of the entities, based on the specified pagination.

IStoryRepository Interface

- **Inherits**: IRepository<Story>
  This interface includes operations related to the Story model, and it adds an additional method:
    - **Task<IEnumerable<Story>> GetFilteredPaginated(string searchKey, int page, int pageSize)**:
        - **Description**: Fetches a paginated list of Story entities filtered by a search key.
        - **Parameters**:
            - searchKey (string): A filter string to search for stories.
            - page (int): The page number to retrieve.
            - pageSize (int): The number of entities per page.

ITeamMemberAvailabilityRepository Interface

- **Inherits**: IRepository<TeamMemberAvailability>
  This interface adds a specific method for fetching TeamMemberAvailability:
    - **Task<TeamMemberAvailability> GetBySprintIdAndUserId(int sprintId, int userId)**:
        - **Description**: Retrieves the availability of a team member for a specific sprint.
        - **Parameters**:
            - sprintId (int): The ID of the sprint.
            - userId (int): The ID of the user.

ITeamRepository Interface

- **Inherits**: IRepository<Team>
  This interface is for operations related to the Team model. It extends the generic repository interface to inherit basic CRUD and pagination methods for Team entities.

IUserRepository Interface

- **Inherits**: IRepository<User>
  This interface includes operations related to the User model and adds two custom methods:
    - **Task<IEnumerable<User>> GetUsersBySprintId(int sprintId)**:
        - **Description**: Retrieves a list of users assigned to a specific sprint.
        - **Parameters**:
            - sprintId (int): The ID of the sprint.
    - **Task<User> GetUserByName(string name)**:
        - **Description**: Retrieves a user by their username.
        - **Parameters**:
            - name (string): The username of the user.

API Controller Documentation - ShiftAvailabilityController

This controller provides endpoints for managing teams and users in the system. It supports operations like retrieving teams, adding random or custom teams, getting team leads, and managing users within teams. The controller interacts with the database using Entity Framework Core and returns responses based on the operation's success or failure.

1. Get All Teams

- **HTTP Method**: GET
- **Endpoint**: /test
- **Description**: Retrieves a list of all teams, including their associated team leads.
- **Response**:
    - 200 OK: Returns a list of teams with their team leads.
    - 400 Bad Request: Returns an error message if the operation fails.

2. Add a Random Team

- **HTTP Method**: POST
- **Endpoint**: /test/AddRandomTeam
- **Description**: Creates and adds a random team with a generated name and description.
- **Response**:
    - 200 OK: Returns the details of the newly created random team.
    - 400 Bad Request: Returns an error message if the operation fails.

3. Get Team Lead for a Specific Team

- **HTTP Method**: GET
- **Endpoint**: /test/GetTeamLead/{teamId}
- **Description**: Retrieves the team lead for a specific team by providing the team ID.
- **Parameters**:
    - teamId (int): The ID of the team whose lead is being requested.
- **Response**:
    - 200 OK: Returns the team lead associated with the given team ID.
    - 400 Bad Request: Returns an error message if the operation fails.

4. Add a Custom Team

- **HTTP Method**: POST
- **Endpoint**: /test/AddTeam
- **Description**: Adds a new team provided in the request body.
- **Request Body**:
    - A Team object containing the team's details.
- **Response**:
    - 200 OK: Returns the details of the newly added team.
    - 400 Bad Request: Returns an error message if the operation fails.

5. Get All Users

- **HTTP Method**: GET
- **Endpoint**: /test/GetAllUsers
- **Description**: Retrieves a list of all users in the system.
- **Response**:
    - 200 OK: Returns a list of all users.
    - 400 Bad Request: Returns an error message if the operation fails.

6. Get Users in a Specific Team

- **HTTP Method**: GET
- **Endpoint**: /test/GetUsersInTeam/{teamId}
- **Description**: Retrieves a list of all users belonging to a specific team, identified by the teamId.
- **Parameters**:

    - teamId (int): The ID of the team for which users are being requested.
- **Response**:
    - 200 OK: Returns a list of users in the specified team.
    - 400 Bad Request: Returns an error message if the operation fails.

In addition to the ShiftAvailabilityController, the application has several other controllers that manage other important pages and functionalities:

1. **BacklogController:**
    - **Manages the backlog of tasks or user stories.**
    - **Provides functionality to view, create, update, and delete tasks in the backlog.**
2. **TicketController:**
    - **Handles operations related to tickets, such as creating, updating, and retrieving tickets for a project or sprint.**
    - **Provides functionality to manage the lifecycle of a ticket, from creation to resolution.**
3. **UserController:**
    - **Manages operations related to users in the system, such as creating, updating, and deleting user records.**
    - **Provides functionality to retrieve user details and assign users to specific teams or sprints.**

Each of these controllers is designed to provide clear and consistent functionality for their respective pages or features in the web application.

Setup:

Prerequisites
Before proceeding, ensure you have the following installed:
- **SQL Server**: Download and install SQL Server [here](https://www.microsoft.com/en-us/sql-server/sql-server-downloads).
- **SQL Server Management Studio (SSMS)**: Download and install SSMS [here](https://learn.microsoft.com/en-us/sql/ssms/download-sql-server-management-studio-ssms).
- **.NET SDK**: Install the latest .NET SDK [here](https://dotnet.microsoft.com/en-us/download).

Step 1: Download and Set Up SQL Server
1. **Download SQL Server**:
   - Visit the [SQL Server download page](https://www.microsoft.com/en-us/sql-server/sql-server-downloads).
   - Choose the edition (Developer or Express is free).

2. **Install SQL Server**:

   - Follow the installation wizard.
   - During setup, choose the "Mixed Mode" authentication option to enable both SQL Server Authentication and Windows Authentication.
   - Note the username (e.g., `sa`) and password you set during installation.

3. **Verify Installation**:
   - Open SQL Server Management Studio (SSMS).
   - Connect to the server using:
     - **Server name**: `localhost` or `localhost\SQLEXPRESS` (depending on your configuration).
     - **Authentication**: SQL Server Authentication or Windows Authentication.

Step 2: Create a Database Using SQL Queries
1. Open **SQL Server Management Studio (SSMS)** and connect to your SQL Server instance.
2. In the query editor, run the following SQL command to create a database named `Jyros`:
   CREATE DATABASE Jyros;

Step 3: Add Connection String in User Secrets
Enable User Secrets for Your Project
1. In the project directory, run:
   dotnet user-secrets init

Add the Connection String
2. Run the following command to set the connection string:
   dotnet user-secrets set "JyrosContext" "Data Source=<server_name>;Initial Catalog=Jyros;Integrated Security=True;TrustServerCertificate=True;"
   Replace the following placeholders:
   - `<server_name>`: The name of your server.

Step 4: Update Local Database Using `dotnet-ef`
Install `dotnet-ef` Tool
1. If not already installed, install the `dotnet-ef` CLI tool:
   ```bash
   dotnet tool install --global dotnet-ef
   ```
Apply Migrations
2. Run the following command to update the local database with the latest migration:
   ```bash
   dotnet ef database update
   ```
Verify Database Changes
3. Open SSMS and connect to your SQL Server instance.
4. Verify that the database has been created and updated with the specified tables and schema.
Step 5: Add Mock Data to the Database
After setting up the database and applying migrations, you can insert mock data to test your application.

AI Model

Architecture

The AI model is developed in **pytorch** and is based on **DistilBERT**, a smaller and optimized version of BERT that retains 97% of its accuracy while being 60% smaller. The model uses pre-trained weights on general language tasks as the base, which gets trained further for estimation of tickets.

Story Point Classification

A classification head was attached to the DISTILBert base to determine story point value for a given task. The model classifies Jira tickets into the following story point categories:

- **1, 2, 3, 5, 8, 13, 14** (where 14 indicates a task too large and needing further breakdown)

Dataset

The dataset used for training comes from JIRA Estimation Prediction Dataset. It includes Jira tickets from 16 open source projects, coming from 9 different organizations. The ticket name and description is the information used to determine story point value. These two fields are concatenated into one single larger text fed to the model.

Training

The **train.py** file provides a simple interface to conduct different experiments and analyze the results. Models can be trained with different settings regarding train-evaluation split of data, number of epochs, batch size and initial learning rate and weight decay. Models can be trained either from the base presented at the beginning or from any checkpoint from previous experiments.

Performance tracking is done through a robust logging system. For each experiment, a folder with the experiment's name gets created. The folder contains a log file that keeps track of loss and learning rate modifications throughout every epoch, as well as **number-of-epochs** subfolders that contain the checkpoint of that epoch and a json file with performance metrics on both the training and validation data (tracked metrics are accuracy, macro and weighted f1 score, precision, recall and the confusion matrix). For ease of reproducing any experiment, the experiment folder also includes a json file with the parameters used to yield these results.

Evaluation

The **eval.py** file provides a method for examining the performance of an existing model in different scenarios. It allows developers to examine performance of a checkpoint on different batches of data from the larger dataset.

Training and evaluation are run on gpu (if available) for better performance.

Model Deployment

The **export.py** file provides a convenient way to save a checkpoint in different formats. Models will be saved both in **safetensors** format (useful for python API's) and **onnx** format (useful for API's in other programming languages). The two models are tested on dummy input before being saved, to assure the model structure was preserved.

Setup

1) **Create a conda environment:**

Ensure you have anaconda or any other conda environment tool installed on your system. Open your terminal and create the environment:

conda create –name <environment_name> python=3.8

Activate the environment:

conda activate <environment_name>

2) **Install dependencies:**

First you need to install the same version of pytorch as the one used by the developer, to avoid possible conflicts. In your terminal run the following command:

pip install torch==2.4.1 –index-url https://download.pytorch.org/whl/cu124

Next, you will need to install the rest of the dependencies. For ease of use, these are all specified in the **requirements.txt** file, along their required version. To install them all, use this command:

pip install -r requirements.txt

3) **Set up environment variables:**

Create a file named **.env** in the root directory of the project and add the following environment variable

DATA_PATH=<path where the dataset files are stored>

By following this guide you should now have a functional repository to experiment with models for story point estimation.

Usage

To see editable training parameters, run the following command:

python train.py –help

To train a model with your desired parameters, run this:

python train.py <parameter_list>

To see evaluation options, run the following command:

python eval.py –help

To evaluate with desired settings, run this:

python eval.py <parameter_list>

To see export options, run the command:

python export.py –help

To export model with desired settings, run this:

python export.py <parameter_list>