LinksPlatform's Platform.Bot Class Library

1.1 ./csharp/Platform.Bot/Program.cs

```
using Interfaces;
using Octokit;
using Platform.Exceptions;
using Platform.IO;
using Storage.Local;
using Storage.Remote.GitHub;
using System;
using System.Collections.Generic;
using System.IO;
using System.Threading;
using Platform.Bot.Trackers;
using Platform.Bot.Triggers;

namespace Platform.Bot
{
    /// <summary>
    /// <para>
    /// Represents the program.
    /// </para>
    /// <para></para>
    /// </summary>
    internal class Program
    {
        private static void Main(string[] args)
        {
            using var cancellation = new ConsoleCancellation();
            var argumentIndex = 0;
            var username = ConsoleHelpers.GetOrReadArgument(argumentIndex++, "Username", args);
            var token = ConsoleHelpers.GetOrReadArgument(argumentIndex++, "Token", args);
            var appName = ConsoleHelpers.GetOrReadArgument(argumentIndex++, "App Name", args);
            var databaseFileName = ConsoleHelpers.GetOrReadArgument(argumentIndex++, "Database
             ↪  file name", args);
            var fileSetName = ConsoleHelpers.GetOrReadArgument(argumentIndex++, "File set name",
             ↪  args);
            var minimumInteractionIntervalStringInputInSeconds =
             ↪  ConsoleHelpers.GetOrReadArgument(argumentIndex, "Minimum interaction interval in
             ↪  seconds", args);
            var minimumInteractionInterval = TimeSpan.FromSeconds(Int32.Parse(minimumInteraction⌋
             ↪  IntervalStringInputInSeconds));
            var dbContext = new FileStorage(databaseFileName);
            Console.WriteLine($"Bot has been started. {Environment.NewLine}Press CTRL+C to
             ↪  close");
            try
            {
                while (true)
                {
                    var api = new GitHubStorage(username, token, appName);
                    var issueTracker = new IssueTracker(api,
                            new HelloWorldTrigger(api, dbContext, fileSetName),
                            new OrganizationLastMonthActivityTrigger(api),
                            new LastCommitActivityTrigger(api),
                            new ProtectMainBranchTrigger(api));
                    var pullRequenstTracker = new PullRequestTracker(api, new
                     ↪  MergeDependabotBumpsTrigger(api));
                    var timestampTracker = new DateTimeTracker(api, new
                     ↪  CreateAndSaveOrganizationRepositoriesMigrationTrigger(api, dbContext,
                     ↪  Directory.GetCurrentDirectory()));
                    issueTracker.Start(cancellation.Token);
                    pullRequenstTracker.Start(cancellation.Token);
                    timestampTracker.Start(cancellation.Token);
                    Thread.Sleep(minimumInteractionInterval);
                }
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex.ToStringWithAllInnerExceptions());
            }
        }
    }
}
```

1.2 ./csharp/Platform.Bot/Trackers/DateTimeTracker.cs

```
using System;
using System.Collections.Generic;
using System.Threading;
using Interfaces;
```

```csharp
using Platform.Timestamps;
using Storage.Remote.GitHub;

namespace Platform.Bot.Trackers;

public class DateTimeTracker : ITracker<DateTime?>
{
    private GitHubStorage _storage { get; }

    private IList<ITrigger<DateTime?>> _triggers { get; }

    public DateTimeTracker(GitHubStorage storage, params ITrigger<DateTime?>[] triggers)
    {
        _storage = storage;
        _triggers = triggers;
    }

    public void Start(CancellationToken cancellationToken)
    {
        foreach (var trigger in _triggers)
        {
            if (cancellationToken.IsCancellationRequested)
            {
                return;
            }
            if (trigger.Condition(null))
            {
                trigger.Action(null);
            }
        }
    }
}
```

1.3 ./csharp/Platform.Bot/Trackers/IssueTracker.cs

```csharp
using Interfaces;
using Octokit;
using Storage.Remote.GitHub;
using System.Collections.Generic;
using System.Threading;

namespace Platform.Bot.Trackers
{
    /// <summary>
    /// <para>
    /// Represents the programmer role.
    /// </para>
    /// <para></para>
    /// </summary>
    public class IssueTracker : ITracker<Issue>
    {
        /// <summary>
        /// <para>
        /// The git hub api.
        /// </para>
        /// <para></para>
        /// </summary>
        private GitHubStorage _storage { get; }

        /// <summary>
        /// <para>
        /// The triggers.
        /// </para>
        /// <para></para>
        /// </summary>
        private IList<ITrigger<Issue>> _triggers { get; }

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="IssueTracker"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="triggers">
        /// <para>A triggers.</para>
        /// <para></para>
        /// </param>
        /// <param name="gitHubApi">
        /// <para>A git hub api.</para>
        /// <para></para>
        /// </param>
        public IssueTracker(GitHubStorage gitHubApi, params ITrigger<Issue>[] triggers)
        {
            _storage = gitHubApi;
            _triggers = triggers;
        }

        /// <summary>
        /// <para>
        /// Starts the cancellation token.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="cancellationToken">
        /// <para>The cancellation token.</para>
        /// <para></para>
        /// </param>
        public void Start(CancellationToken cancellationToken)
        {
            foreach (var trigger in _triggers)
            {
                foreach (var issue in _storage.GetIssues())
                {
                    if (cancellationToken.IsCancellationRequested)
                    {
                        return;
                    }
                    if (trigger.Condition(issue))
                    {
                        trigger.Action(issue);
                    }
                }
            }
        }
    }
}
```

1.4 ./csharp/Platform.Bot/Trackers/PullRequestTracker.cs

```
using Interfaces;
using Octokit;
using Storage.Remote.GitHub;
using System.Collections.Generic;
using System.Threading;
using Platform.Collections.Lists;
using Platform.Threading;

namespace Platform.Bot.Trackers
{
    /// <summary>
    /// <para>
    /// Represents the programmer role.
    /// </para>
    /// <para></para>
    /// </summary>
    public class PullRequestTracker : ITracker<PullRequest>
    {
        /// <summary>
        /// <para>
        /// The git hub api.
        /// </para>
        /// <para></para>
        /// </summary>
        private GitHubStorage _storage;

        /// <summary>
        /// <para>
        /// The triggers.
        /// </para>
        /// <para></para>
        /// </summary>
        private IList<ITrigger<PullRequest>> _triggers;

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="IssueTracker"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="triggers">
```

```csharp
        /// <para>A triggers.</para>
        /// <para></para>
        /// </param>
        /// <param name="storage">
        /// <para>A git hub api.</para>
        /// <para></para>
        /// </param>
        public PullRequestTracker(GitHubStorage storage, params ITrigger<PullRequest>[] triggers)
        {
            _storage = storage;
            _triggers = triggers;
        }

        /// <summary>
        /// <para>
        /// Starts the cancellation token.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="cancellationToken">
        /// <para>The cancellation token.</para>
        /// <para></para>
        /// </param>
        public void Start(CancellationToken cancellationToken)
        {
            foreach (var trigger in _triggers)
            {
                foreach (var repository in
                ↪  _storage.Client.Repository.GetAllForOrg("linksplatform").AwaitResult())
                {
                    foreach (var pullRequest in _storage.Client.PullRequest.GetAllForRepository(⌟
                    ↪  repository.Id).AwaitResult())
                    {
                        if (cancellationToken.IsCancellationRequested)
                        {
                            return;
                        }
                        var detailedPullRequest = _storage.Client.PullRequest.Get(repository.Id,
                        ↪  pullRequest.Number).AwaitResult();
                        if (trigger.Condition(detailedPullRequest))
                        {
                            trigger.Action(detailedPullRequest);
                        }
                    }
                }
            }
        }
    }
}
```

1.5 ./csharp/Platform.Bot/Triggers/Activity.cs

```csharp
using System.Collections.Generic;

namespace Platform.Bot.Triggers
{
    internal class Activity
    {
        public string Url { get; set; }

        public List<string> Repositories { get; set; }
    }
}
```

1.6 ./csharp/Platform.Bot/Triggers/CreateAndSaveOrganizationRepositoriesMigrationTrigger.cs

```csharp
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Interfaces;
using Octokit;
using Platform.Data;
using Platform.Data.Doublets;
using Platform.Timestamps;
using Storage.Local;
using Storage.Remote.GitHub;

namespace Platform.Bot.Triggers;

```

```
public class CreateAndSaveOrganizationRepositoriesMigrationTrigger : ITrigger<DateTime?>
{
    private readonly GitHubStorage _githubStorage;

    private readonly FileStorage _linksStorage;

    private string _filePath;

    public CreateAndSaveOrganizationRepositoriesMigrationTrigger(GitHubStorage githubStorage,
    ↪  FileStorage linksStorage, string filePath)
    {
        _githubStorage = githubStorage;
        _linksStorage = linksStorage;
        _filePath = filePath;

    }
    public bool Condition(DateTime? dateTime)
    {
        var allMigrations = _githubStorage.GetAllMigrations("linksplatform");
        if (allMigrations.Count == 0)
        {
            return true;
        }
        var lastMigrationTimestamp = Convert.ToDateTime(allMigrations.Last().CreatedAt);
        var timeAfterLastMigration = DateTime.Now - lastMigrationTimestamp;
        return timeAfterLastMigration.Days > 1;
    }

    public async void Action(DateTime? dateTime)
    {
        var repositoryNames = _githubStorage.GetAllRepositoryNames("linksplatform");
        var createMigrationResult = await _githubStorage.CreateMigration("linksplatform",
        ↪  repositoryNames);
        if (null == createMigrationResult || createMigrationResult.State.Value ==
        ↪  Migration.MigrationState.Failed)
        {
            Console.WriteLine("Migration is failed.");
            return;
        }
        Console.WriteLine($"Saving migration {createMigrationResult.Id}.");
        await _githubStorage.SaveMigrationArchive("linksplatform", createMigrationResult.Id,
        ↪  _filePath);
        Console.WriteLine($"Migration {createMigrationResult.Id} is saved.");
    }
}
```

1.7 ./csharp/Platform.Bot/Triggers/HelloWorldTrigger.cs

```
using Interfaces;
using Octokit;
using Storage.Local;
using Storage.Remote.GitHub;

namespace Platform.Bot.Triggers
{
    using TContext = Issue;
    /// <summary>
    /// <para>
    /// Represents the hello world trigger.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="ITrigger{TContext}"/>
    internal class HelloWorldTrigger : ITrigger<TContext>
    {
        private readonly GitHubStorage _storage;
        private readonly FileStorage _fileStorage;
        private readonly string _fileSetName;

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="HelloWorldTrigger"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="storage">
        /// <para>A git hub api.</para>
        /// <para></para>
        /// </param>
        /// <param name="fileStorage">
        /// <para>A file storage.</para>
        /// <para></para>
        /// </param>
        /// <param name="fileSetName">
        /// <para>A file set name.</para>
        /// <para></para>
        /// </param>
        public HelloWorldTrigger(GitHubStorage storage, FileStorage fileStorage, string
         ↪  fileSetName)
        {
            this._storage = storage;
            this._fileStorage = fileStorage;
            this._fileSetName = fileSetName;
        }


        /// <summary>
        /// <para>
        /// Actions the context.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="context">
        /// <para>The context.</para>
        /// <para></para>
        /// </param>

        public void Action(TContext context)
        {
            foreach (var file in _fileStorage.GetFilesFromSet(_fileSetName))
            {
                _storage.CreateOrUpdateFile(context.Repository.Name,
                 ↪  context.Repository.DefaultBranch, file);
            }
            _storage.CloseIssue(context);
        }


        /// <summary>
        /// <para>
        /// Determines whether this instance condition.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="context">
        /// <para>The context.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        public bool Condition(TContext context) => context.Title.ToLower() == "hello world";
    }
}
```

1.8 ./csharp/Platform.Bot/Triggers/LastCommitActivityTrigger.cs

```
using Interfaces;
using Octokit;
using Platform.Communication.Protocol.Lino;
using Storage.Remote.GitHub;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Platform.Bot.Triggers
{
    using TContext = Issue;
    internal class LastCommitActivityTrigger : ITrigger<TContext>
    {
        private readonly GitHubStorage _storage;

        private readonly Parser _parser = new();

        public LastCommitActivityTrigger(GitHubStorage storage) => _storage = storage;

        public bool Condition(TContext context) => context.Title.ToLower() == "last 3 months
         ↪  commit activity";

        public void Action(TContext context)
        {
            var issueService = _storage.Client.Issue;
            var owner = context.Repository.Owner.Login;
            var ignoredRepositories =
                context.Body != null ? GetIgnoredRepositories(_parser.Parse(context.Body)) : default;
            var users = GetActivities(owner, ignoredRepositories);
            StringBuilder sb = new();
            foreach (var user in users)
            {
                sb.AppendLine($"- **{user.Url.Replace("api.", "").Replace("users/", "")}**");
                // Break line
                sb.AppendLine("------------------");
            }
            var result = sb.ToString();
            var comment = issueService.Comment.Create(owner, context.Repository.Name, context.Number, result);
            comment.Wait();
            Console.WriteLine($"Last commit activity comment is added: {comment.Result.HtmlUrl}");
            _storage.CloseIssue(context);
        }

        public HashSet<string> GetIgnoredRepositories(IList<Link> links)
        {
            HashSet<string> ignoredRepos = new() { };
            foreach (var link in links)
            {
                var values = link.Values;
                if (values != null && values.Count == 3 && string.Equals(values.First().Id, "ignore", StringComparison.OrdinalIgnoreCase) && string.Equals(values.Last().Id.Trim('.'), "repository", StringComparison.OrdinalIgnoreCase))
                {
                    ignoredRepos.Add(values[1].Id);
                }
            }
            return ignoredRepos;
        }

        public HashSet<Activity> GetActivities(string owner, HashSet<string> ignoredRepositories = default)
        {
            HashSet<Activity> activeUsers = new();
            foreach (var repository in _storage.Client.Repository.GetAllForOrg(owner).Result)
            {
                if (ignoredRepositories?.Contains(repository.Name) ?? false)
                {
                    continue;
                }
                foreach (var commit in _storage.GetCommits(repository.Owner.Login, repository.Name, DateTime.Today.AddMonths(-3)))
                {
                    if (!_storage.Client.Organization.Member.CheckMember(owner, commit.Author.Login).Result)
                    {
                        continue;
                    }
                    if (!activeUsers.Any(x => x.Url == commit.Author.Url))
                    {
                        activeUsers.Add(new Activity() { Url = commit.Author.Url, Repositories = new List<string> { repository.Url } });
                    }
                    else
                    {
                        if (!activeUsers.Any(x => x.Repositories.Any(y => y == repository.Url) == true))
                        {
                            activeUsers.FirstOrDefault(x => x.Url == commit.Author.Url).Repositories.Add(repository.Url);
                        }
                    }
                }
            }
            return activeUsers;
        }
```

```
    }
}
```

1.9 ./csharp/Platform.Bot/Triggers/MergeDependabotBumpsTrigger.cs

```
using System;
using Interfaces;
using Octokit;
using Platform.Threading;
using Storage.Remote.GitHub;

namespace Platform.Bot.Triggers
{
    public class MergeDependabotBumpsTrigger : ITrigger<PullRequest>
    {
        private const int DependabotId = 49699333;
        private readonly GitHubStorage _githubStorage;
        public MergeDependabotBumpsTrigger(GitHubStorage storage)
        {
            _githubStorage = storage;
        }

        public bool Condition(PullRequest pullRequest)
        {
            var isDependabotAuthor = DependabotId == pullRequest.User.Id;
            if (!isDependabotAuthor)
            {
                return false;
            }
            var isTestAndDeployCompleted = false;
            var repositoryId = pullRequest.Base.Repository.Id;
            var checks = _githubStorage.Client.Check.Run.GetAllForReference(repositoryId,
            ↪  pullRequest.Head.Sha).AwaitResult();
            foreach (var checkRun in checks.CheckRuns)
            {
                if (checkRun.Name is "testAndDeploy" or "deploy" && checkRun.Status.Value ==
                ↪  CheckStatus.Completed)
                {
                    isTestAndDeployCompleted = true;
                }
            }
            var isMergable = pullRequest.Mergeable ?? false;
            return isTestAndDeployCompleted && isMergable;
        }

        public void Action(PullRequest pullRequest)
        {
            var repositoryId = pullRequest.Base.Repository.Id;
            var prMerge = _githubStorage.Client.PullRequest.Merge(repositoryId,
            ↪  pullRequest.Number, new MergePullRequest()).AwaitResult();
            Console.WriteLine($"{pullRequest.HtmlUrl} is {(prMerge.Merged ? "successfully":"not
            ↪  successfully")} merged.");
        }
    }
}
```

1.10 ./csharp/Platform.Bot/Triggers/OrganizationLastMonthActivityTrigger.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using Interfaces;
using Octokit;
using Platform.Communication.Protocol.Lino;
using Storage.Remote.GitHub;

namespace Platform.Bot.Triggers
{
    using TContext = Issue;
    /// <summary>
    /// <para>
    /// Represents the organization last month activity trigger.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="ITrigger{Issue}"/>
    internal class OrganizationLastMonthActivityTrigger : ITrigger<TContext>
    {
        private readonly GitHubStorage _storage;
        private readonly Parser _parser = new();

        /// <summary>
```

```csharp
        /// <para>
        /// Initializes a new <see cref="OrganizationLastMonthActivityTrigger"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="storage">
        /// <para>A storage.</para>
        /// <para></para>
        /// </param>
        public OrganizationLastMonthActivityTrigger(GitHubStorage storage) => _storage = storage;

        /// <summary>
        /// <para>
        /// Determines whether this instance condition.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="context">
        /// <para>The context.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        public bool Condition(TContext context) => context.Title.ToLower() == "organization last month activity";

        /// <summary>
        /// <para>
        /// Actions the context.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="context">
        /// <para>The context.</para>
        /// <para></para>
        /// </param>
        public void Action(TContext context)
        {
            var issueService = _storage.Client.Issue;
            var owner = context.Repository.Owner.Login;
            var activeUsersString = string.Join("\n", GetActiveUsers(GetIgnoredRepositories(_parser.Parse(context.Body)), owner));
            issueService.Comment.Create(owner, context.Repository.Name, context.Number, activeUsersString);
            _storage.CloseIssue(context);
        }

        /// <summary>
        /// <para>
        /// Gets the ignored repositories using the specified links.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>The links.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The ignored repos.</para>
        /// <para></para>
        /// </returns>
        public HashSet<string> GetIgnoredRepositories(IList<Link> links)
        {
            HashSet<string> ignoredRepos = new() { };
            foreach (var link in links)
            {
                var values = link.Values;
                if (values != null && values.Count == 3 && string.Equals(values.First().Id, "ignore", StringComparison.OrdinalIgnoreCase) && string.Equals(values.Last().Id.Trim('.'), "repository", StringComparison.OrdinalIgnoreCase))
                {
                    ignoredRepos.Add(values[1].Id);
                }
            }
            return ignoredRepos;
        }

        /// <summary>
        /// <para>
        /// Gets the active users using the specified ignored repositories.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="ignoredRepositories">
        /// <para>The ignored repositories.</para>
        /// <para></para>
        /// </param>
        /// <param name="owner">
        /// <para>The owner.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The active users.</para>
        /// <para></para>
        /// </returns>
        public HashSet<string> GetActiveUsers(HashSet<string> ignoredRepositories, string owner)
        {
            HashSet<string> activeUsers = new();
            var date = DateTime.Now.AddMonths(-1);
            foreach (var repository in _storage.Client.Repository.GetAllForOrg(owner).Result)
            {
                if (ignoredRepositories.Contains(repository.Name))
                {
                    continue;
                }
                foreach (var commit in _storage.GetCommits(repository.Owner.Login,
                ↪  repository.Name))
                {

                    activeUsers.Add(commit.Author.Login);

                }
                foreach (var pullRequest in _storage.GetPullRequests(repository.Owner.Login,
                ↪  repository.Name))
                {
                    foreach (var reviewer in pullRequest.RequestedReviewers)
                    {
                        if (pullRequest.CreatedAt < date || pullRequest.UpdatedAt < date ||
                        ↪  pullRequest.ClosedAt < date || pullRequest.MergedAt < date)
                        {
                            activeUsers.Add(reviewer.Login);
                        }
                    }
                }
                foreach (var createdIssue in _storage.GetIssues(repository.Owner.Login,
                ↪  repository.Name))
                {
                    activeUsers.Add(createdIssue.User.Login);
                }
            }
            return activeUsers;
        }
    }
}
```

1.11 ./csharp/Platform.Bot/Triggers/ProtectMainBranchTrigger.cs

```csharp
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Interfaces;
using Octokit;
using Storage.Remote.GitHub;

namespace Platform.Bot.Triggers
{
    using TContext = Issue;
    public class ProtectMainBranchTrigger : ITrigger<TContext>
    {
        private readonly GitHubStorage _storage;
        public ProtectMainBranchTrigger(GitHubStorage storage) => _storage = storage;
        public bool Condition(TContext context) => context.Title.ToLower() == "protect default
        ↪  branch in all organization's repositories";

        public void Action(TContext context)
        {
            var repositories = _storage.Client.Repository.GetAllForOrg(context.Repository.Owner.Login).Result;
            var results = UpdateRepositoriesDefaultBranchProtection(repositories);
            StringBuilder failedRepositoriesComment = new(repositories.Count * repositories[0].Name.Length);
            foreach (var result in results.Where(result => !result.Value))
            {
                failedRepositoriesComment.AppendLine($"- [ ] {result.Key}");
            }
            if (failedRepositoriesComment.Length != 0)
            {
                failedRepositoriesComment.AppendLine(
                    "TODO: Fix default branch protection of these repositories. Failed repositories:");
                _storage.Client.Issue.Comment.Create(context.Repository.Id, context.Number, failedRepositoriesComment.ToString());
            }
            else
            {
                _storage.Client.Issue.Comment.Create(context.Repository.Id, context.Number, "Success. All repositories default branch protection is updated.");
                _storage.CloseIssue(context);
            }
        }

        public Dictionary<string, bool> UpdateRepositoriesDefaultBranchProtection(IReadOnlyList<Repository> repositories)
        {
            Dictionary<string, bool> result = new(repositories.Count);
            foreach (var repository in repositories)
            {
                if (repository.Private)
                {
                    continue;
                }
                var update = new BranchProtectionSettingsUpdate(new BranchProtectionPushRestrictionsUpdate());
                var request =
                    _storage.Client.Repository.Branch.UpdateBranchProtection(repository.Id,
                        repository.DefaultBranch, update);
                request.Wait();
                result.Add(repository.Name, request.IsCompletedSuccessfully);
            }
            return result;
        }
    }
}
```

Index