Project 2 – API-Based Systems Integration

Option 1 – User Dashboard for GitHub Projects

This blueprint gives you: SCRUM setup, architecture, OAuth flow, GitHub API usage (REST + GraphQL), data model, metrics logic (including lines added/removed), UI, and documentation templates (UML + BPMN). It's designed for fast delivery with clear sprints.

---

1) SCRUM Setup (lightweight)

- **Cadence:** 1–2 week sprints, daily standup (≤15 min).
- **Roles:** Product Owner (faculty/client), Scrum Master, Dev Team.
- **Boards:** Trello (or GitHub Projects) with columns: Backlog → Ready → In Progress → Code Review → Done.
- **Ceremonies:** Sprint Planning, Review (demo), Retro. Definition of Done includes tests, docs, and deployment.
- **Artifacts:** Product Backlog, Sprint Backlog, Burndown chart (export from Trello/GitHub), Increment (deployed web app).

Trello board starter lists - Icebox, Product Backlog, Sprint Backlog, In Progress, In Review, Testing, Done. - Labels: `frontend`, `backend`, `devops`, `docs`, `good-first`, `blocked`.

---

2) Tech Stack

- **Frontend:** React + Vite or Next.js; State: React Query/Zustand; Charts: Recharts; UI: Tailwind + shadcn/ui.
- **Backend:** Spring Boot (Java) or Node/Express (choose one). Sample specs below assume **Spring Boot**.
- **DB/Cache:** PostgreSQL + Redis (optional) for caching GitHub responses and rate-limit protection.
- **Auth:** GitHub OAuth App (PKCE for SPA or server-side OAuth code flow).
- **Hosting:** Vercel/Netlify (frontend), Render/Fly.io/Heroku (backend), Neon/Supabase (Postgres).
- **CI:** GitHub Actions (build, test, lint, deploy).

---

3) GitHub OAuth – Flows & Scopes

- **Public repos only:** `read:user`, `user:email`.
- **Private repos:** add `repo` (or `repo:status`/`repo_deployment` granular scopes as needed).

Server-side Code Flow (recommended) 1. Client → `/auth/login` → redirect to GitHub authorize endpoint. 2. GitHub → callback `/auth/callback?code=...`. 3. Backend exchanges `code` for **access token**; store encrypted in DB/session. 4. Issue your own **JWT session** to frontend.

Security: PKCE, HTTPS only, SameSite cookies, rotate tokens, never expose GitHub token to the browser when possible.

4) Repository Selection UX

- Input: paste repo URL `https://github.com/{owner}/{repo}` **or** browse authenticated user's repos via API.
- Persist last selections per user.

5) Data Sources & Key Endpoints

Use **GraphQL** where aggregation helps, **REST** where simple.

REST (selected)

- List contributors: `GET /repos/{owner}/{repo}/contributors` (fallback; may miss non-default branch committers).
- Commits by author: `GET /repos/{owner}/{repo}/commits?author={login}&since=&until=` (paginate).
- Single commit stats (additions/deletions): `GET /repos/{owner}/{repo}/commits/{sha}`.
- Issues: `GET /repos/{owner}/{repo}/issues?state=all&since=` (filter by `creator`, `assignee` for per-user).
- Pull Requests: `GET /repos/{owner}/{repo}/pulls?state=all` and `GET /repos/{owner}/{repo}/pulls/{number}` (for `additions`, `deletions`, `merged_by`).
- Events (recent activity): `GET /repos/{owner}/{repo}/events` and user `GET /users/{login}/events/public`.

GraphQL (selected)

- **Per-user, per-repo contributions**

```
query($owner: String!, $name: String!, $login: String!, $from:
DateTime, $to: DateTime) {
  repository(owner: $owner, name: $name) {
    defaultBranchRef { name }
    pullRequests(first: 100, states: [OPEN, MERGED, CLOSED], orderBy:
{field:UPDATED_AT, direction:DESC}) {
      nodes { author { login } additions deletions merged at createdAt
state }
    }
    issues(first: 100, orderBy:{field:UPDATED_AT, direction:DESC}) {
      nodes { author { login } closed at createdAt state }
    }
    object(expression: "HEAD") { ... on Commit {
      history(first: 100, author: { id: null, emails: null, idIn: null,
login: $login }, since: $from, until: $to) {
        totalCount
        edges { node { oid additions deletions committedDate message } }
      }
    }}
  }
}
```

- Use cursors to paginate until coverage is sufficient for the date range.

Rate limits: honor `X-RateLimit-Remaining`, use ETags (`If-None-Match`), backoff + caching.

---

6) Metrics Logic (per contributor)

a) # Commits - Count commits authored (email/login match). Use GraphQL `history` or REST commits filtered by `author`.

b) Lines added/removed - Sum `additions`/`deletions` across **PRs merged** where the contributor is the author, OR across **commits** if you need branch-level accuracy. Prefer PR-based for reviewable units; commit-based if team often pushes directly.

c) Issues opened/closed - `author.login == contributor`; `closedBy` if needed. Filter by dates.

d) PRs submitted and approved - Submitted: PRs where `author.login == contributor`. - Approved: use Reviews: `GET /pulls/{number}/reviews` (REST) or GraphQL `reviews(states: [APPROVED])` where `author.login` equals approver.

e) Recent activity - Blend last N events for contributor: `PushEvent`, `PullRequestEvent`, `IssuesEvent`, `ReviewEvent`, etc. Render a timeline.

Time windows - Default: last 30/90 days with date pickers; allow "All time" (may be slow – use background job & cache).

---

7) System Architecture

```
[React SPA/Next.js] ──JWT──► [API Gateway (Spring Boot)] ──► [GitHub REST/
GraphQL]
                                          │
                                          ├──► [PostgreSQL] (users, repos,
snapshots, caches)
                                          └──► [Redis] (rate-limit, ETag cache,
sessions)
```

Backend Modules - `auth`: OAuth, token storage (encrypted), session JWTs. - `github`: clients (REST/GraphQL), pagination helpers, ETag caching. - `metrics`: aggregation by contributor, time windows. - `api`: REST endpoints for frontend.

Frontend Modules - `auth`: login, callback handler. - `repo-picker`: paste URL or browse repos. - `dashboard`: per-contributor pages + team overview. - `charts`: reusable chart components.

---

8) API Design (your backend)

- `POST /auth/login` → redirect URL
- `GET /auth/callback` → sets session
- `GET /api/repos?mine=true&visibility=all` → list repos
- `GET /api/repos/parse?url=...` → owner/repo resolve + access check
- `GET /api/repos/{owner}/{repo}/contributors` → list w/ basic stats
- `GET /api/repos/{owner}/{repo}/contributors/{login}/metrics?from=&to=` → full metrics
- `GET /api/repos/{owner}/{repo}/activity?login=&limit=50` → recent activity items
- `POST /api/snapshots` (optional) → persist snapshot for reports

Responses are small JSON documents designed for the UI; large raw GitHub payloads are never forwarded to the client.

---

9) Data Model (Postgres)

- `users(id, login, name, avatar_url, email, github_token_enc, created_at)`
- `repos(id, owner, name, url, is_private)`
- `user_repo_access(user_id, repo_id, scope)`
- `metrics_snapshot(id, repo_id, taken_at, window_from, window_to)`
- `contrib_metrics(snapshot_id, login, commits, additions, deletions, issues_opened, issues_closed, prs_submitted, prs_approved)`
- `etag_cache(key, etag, last_json, updated_at)`

---

10) UI/UX – Wireframe Notes

- **Home:** login with GitHub → repo picker.
- **Repo Overview:** contributors table w/ sortable columns; quick filters by date range.
- **Contributor Dashboard:**
- KPI cards: Commits, Added, Removed, Issues Opened/Closed, PRs Submitted/Approved.
- Charts: Commits over time (line), Additions/Deletions (stacked bar), Issue/PR breakdown (pie/donut).
- Recent Activity timeline.
- **Responsive:** grid → single column on mobile; keyboard nav; accessible (ARIA, semantic HTML).

---

11) UML – C4 + Class Diagram (PlantUML)

C4 Context (simplified)

```
@startuml
Person(user, "Contributor/User")
System_Boundary(sys, "GitHub Dashboard") {
  Container(web, "Web App", "React")
  Container(api, "API Server", "Spring Boot")
  ContainerDb(db, "PostgreSQL", "Metrics & tokens (enc)")
}
System_Ext(gh, "GitHub API", "REST/GraphQL")
Rel(user, web, "Uses via browser")
Rel(web, api, "JWT HTTPS")
Rel(api, db, "JDBC")
Rel(api, gh, "OAuth token calls")
@enduml
```

Backend Class Diagram (key parts)

```
@startuml
class OAuthService { +getLoginUrl(); +handleCallback(code) }
class GitHubClient { +restGet(); +graphql(); }
class MetricsService { +compute(repo, login, from, to): ContributorMetrics }
class ContributorMetrics { commits; additions; deletions; issuesOpened;
issuesClosed; prsSubmitted; prsApproved }
class CacheStore { +getETag(key); +putETag(key, etag, json) }
OAuthService -> GitHubClient
MetricsService -> GitHubClient
MetricsService -> CacheStore
@enduml
```

---

12) BPMN – "Generate Contributor Dashboard" (PlantUML)

```
@startuml
' BPMN with PlantUML
!include <bpmn>
start
:User selects repo;
:Check access via GitHub token;
if (Has access?) then (yes)
  :Fetch contributors;
  fork
    :Fetch commits per contributor;
  fork again
    :Fetch PRs & reviews;
  fork again
    :Fetch issues;
  end fork
  :Aggregate metrics;
  :Cache results + store snapshot;
  :Render charts/UI;
else (no)
  :Show error / request broader scope;
endif
```

```
stop
@enduml
```

---

13) Testing & Quality

- **Unit:** services, GitHub client adapters (mock HTTP).
- **Integration:** hit GitHub sandbox via recorded cassettes (WireMock).
- **E2E:** Playwright/Cypress for UI flows.
- **Load:** small soak tests to respect rate limits.

---

14) DevOps

- **Secrets:** store GitHub client secret in platform secrets manager.
- **CI:** build → test → lint → Docker → deploy. Tag `main` → production.
- **Monitoring:** basic logs + uptime; alert on 401/403 spikes (expired tokens/scopes).

---

15) Sprint Plan (example 2 sprints)

Sprint 1 (Auth & MVP) - GitHub OAuth login + callback - Repo picker (browse + paste URL) - Contributors list (names/avatars) - Compute commits + PR count per contributor (last 30d)

Sprint 2 (Full Metrics + UI) - Additions/deletions, issues opened/closed, approvals - Activity timeline - Charts & responsive layout - Snapshots + export (CSV/PNG)

---

16) GitHub & Documentation

- **Artifacts in GitHub:** code, ADRs, UML/BPMN diagrams, API docs (OpenAPI), test reports, CI configs.
- **README:** architecture, setup, scopes, env vars, run/dev scripts.
- **OpenAPI** for your backend endpoints (generated from Spring / Swagger).
- **BPMN/UML**: keep `.puml` files in `/docs/` and PNG exports.

---

17) Example DTOs & Endpoints (Spring Boot)

```java
// Contributor metrics response
public record ContributorMetricsDto(
  String login, String avatarUrl,
  int commits, int additions, int deletions,
  int issuesOpened, int issuesClosed,
  int prsSubmitted, int prsApproved,
  List<ActivityItem> recent
) {}
```

```
@RestController
@RequestMapping("/api/repos/{owner}/{repo}")
class MetricsController {
  @GetMapping("/contributors/{login}/metrics")
  ContributorMetricsDto metrics(
    @PathVariable String owner,
    @PathVariable String repo,
    @PathVariable String login,
    @RequestParam Optional<Instant> from,
    @RequestParam Optional<Instant> to) { /* ... */ }
}
```

---

18) Risks & Mitigations

- **Rate Limits:** cache & snapshot; allow manual refresh.
- **Token Scope Insufficient:** detect 403; prompt to re-auth with broader scope.
- **Large Repos:** async background aggregation + progress UI.
- **Identity Matching:** prefer GitHub login over emails; handle bots.

---

19) Acceptance Criteria (Definition of Done)

- OAuth login works; users can browse/select repos.
- For chosen repo, each contributor has a dashboard with the listed metrics.
- Charts render on desktop & mobile; performance acceptable (<2s cached views).
- Project artifacts on GitHub; board on Trello/GitHub Projects.
- UML & BPMN diagrams checked into `/docs/`.

---

20) Nice-to-have

- Export to CSV/PNG/PDF.
- Team heatmap calendar; velocity chart by sprint (labels on PRs).
- Notifications for review requests.