

# gitprint

---

<b>Branch</b>	main
<b>Commit</b>	699fff0
<b>Author</b>	Izel Nakri <contact@izelnakri.com>
<b>Date</b>	2026-03-14 23:51:02 +0100
<b>Message</b>	dev: now we lint for docs & doctests
<b>Files</b>	43
<b>Lines</b>	10871
<b>Repo Size</b>	1.8 MB
<b>FS Size</b>	708.9 MB
<b>FS Owner</b>	runner
<b>FS Group</b>	runner
<b>Generated</b>	2026-03-14 22:54:27 UTC

---

# Table of Contents

.claude/commands/doctest.md	p.4	8 LOC	·	636 B	·	2026-03-14
.claude/settings.local.json	p.5	27 LOC	·	630 B	·	2026-03-02
.dockerignore	p.6	5 LOC	·	45 B	·	2026-03-02
.github/workflows/ci.yml	p.7	251 LOC	·	9.8 KB	·	2026-03-14
.github/workflows/cleanup.yml	p.11	31 LOC	·	1.1 KB	·	2026-03-04
.github/workflows/release.yml	p.12	195 LOC	·	6.9 KB	·	2026-03-04
.gitignore	p.15	5 LOC	·	43 B	·	2026-02-19
CHANGELOG.md	p.16	169 LOC	·	11.5 KB	·	2026-03-04
CLAUDE.md	p.19	47 LOC	·	2.0 KB	·	2026-03-04
Cargo.toml	p.20	52 LOC	·	1.8 KB	·	2026-03-04
Dockerfile	p.21	41 LOC	·	1.7 KB	·	2026-03-04
Makefile	p.22	100 LOC	·	4.4 KB	·	2026-03-14
README.md	p.24	241 LOC	·	7.7 KB	·	2026-03-14
benches/pipeline.rs	p.28	91 LOC	·	2.4 KB	·	2026-02-18
build.rs	p.30	23 LOC	·	806 B	·	2026-03-02
cliff.toml	p.31	64 LOC	·	2.8 KB	·	2026-03-02
demo/demo.tape	p.32	56 LOC	·	890 B	·	2026-03-05
flake.nix	p.33	117 LOC	·	3.6 KB	·	2026-03-14
rust-toolchain.toml	p.35	3 LOC	·	88 B	·	2026-03-14
scripts/check_benchmarks.py	p.36	101 LOC	·	3.3 KB	·	2026-02-19
src/cli.rs	p.38	392 LOC	·	12.8 KB	·	2026-03-14
src/defaults.rs	p.44	135 LOC	·	3.1 KB	·	2026-03-14
src/filter.rs	p.46	326 LOC	·	10.4 KB	·	2026-02-18
src/git.rs	p.51	858 LOC	·	27.6 KB	·	2026-03-14
src/github.rs	p.63	435 LOC	·	14.1 KB	·	2026-03-14
src/highlight.rs	p.69	242 LOC	·	7.4 KB	·	2026-02-18
src/lib.rs	p.73	525 LOC	·	17.8 KB	·	2026-03-14
src/main.rs	p.80	383 LOC	·	11.8 KB	·	2026-03-05
src/pdf/code.rs	p.85	208 LOC	·	6.3 KB	·	2026-03-14
src/pdf/cover.rs	p.88	409 LOC	·	15.2 KB	·	2026-03-14
src/pdf/diff.rs	p.94	288 LOC	·	10.2 KB	·	2026-03-14
src/pdf/fonts.rs	p.98	47 LOC	·	1.5 KB	·	2026-03-14
src/pdf/layout.rs	p.99	775 LOC	·	25.3 KB	·	2026-03-14
src/pdf/mod.rs	p.109	135 LOC	·	4.3 KB	·	2026-03-14
src/pdf/toc.rs	p.111	194 LOC	·	6.2 KB	·	2026-03-14
src/pdf/tree.rs	p.114	154 LOC	·	4.3 KB	·	2026-03-14
src/pdf/user_activity.rs	p.116	688 LOC	·	24.6 KB	·	2026-03-14
src/pdf/user_cover.rs	p.125	262 LOC	·	9.4 KB	·	2026-03-14
src/pdf/user_repos.rs	p.129	531 LOC	·	19.0 KB	·	2026-03-14
src/preview.rs	p.136	1032 LOC	·	34.4 KB	·	2026-03-04
src/types.rs	p.150	239 LOC	·	7.4 KB	·	2026-03-14
src/user_report.rs	p.154	552 LOC	·	18.8 KB	·	2026-03-14
tests/integration.rs	p.162	434 LOC	·	14.6 KB	·	2026-02-18

- 3  
**File Tree**

```
.claude
├── commands
│   └── doctest.md
├── settings.local.json
├── .dockerignore
├── .github
│   └── workflows
│       ├── ci.yml
│       ├── cleanup.yml
│       └── release.yml
├── .gitignore
├── CHANGELOG.md
├── CLAUDE.md
├── Cargo.toml
├── Dockerfile
├── Makefile
├── README.md
├── benches
│   └── pipeline.rs
├── build.rs
├── cliff.toml
├── demo
│   └── demo.tape
├── flake.nix
├── rust-toolchain.toml
├── scripts
│   └── check_benchmarks.py
├── src
│   ├── cli.rs
│   ├── defaults.rs
│   ├── filter.rs
│   ├── git.rs
│   ├── github.rs
│   ├── highlight.rs
│   ├── lib.rs
│   ├── main.rs
│   ├── pdf
│   │   ├── code.rs
│   │   ├── cover.rs
│   │   ├── diff.rs
│   │   ├── fonts.rs
│   │   ├── layout.rs
│   │   ├── mod.rs
│   │   ├── toc.rs
│   │   ├── tree.rs
│   │   ├── user_activity.rs
│   │   ├── user_cover.rs
│   │   └── user_repos.rs
│   ├── preview.rs
│   ├── types.rs
│   └── user_report.rs
├── tests
│   └── integration.rs
```

## .claude/commands/doctest.md

1 # doctest

2

3 When writing or modifying public functions, methods, or types in this codebase:

4

5 1. Every `pub fn`, `pub async fn`, and `pub` method must have a `///` doc comment describing what it does.

6 2. Every such doc comment must include a `# Examples` section with a working `rust` doctest that compile

7 3. When editing an existing public item that already has a doctest, keep it – update it only if the signature or be

8 4. Doctests must be self-contained: import everything they need, construct any required inputs inline, and assert a

### .claude/settings.local.json

```
1 {
2   "permissions": {
3     "allow": [
4       "WebFetch(domain:docs.rs)",
5       "WebSearch",
6       "WebFetch(domain:github.com)",
7       "WebFetch(domain:crates.io)",
8       "Bash(cargo search:*)",
9       "Bash(cargo test:*)",
10      "Bash(cargo clippy:*)",
11      "Bash(cargo fmt:*)",
12      "Bash(nix flake check:*)",
13      "Bash(git status:*)",
14      "Bash(git diff:*)",
15      "Bash(git log:*)",
16      "Bash(git add:*)",
17      "Bash(gh run list:*)",
18      "Bash(echo:*)",
19      "Bash(cargo doc:*)",
20      "Bash(wc:*)",
21      "Bash(grep:*)",
22      "Bash(cargo tree:*)",
23      "Bash(make check:*)",
24      "Bash(cargo check:*)"
25    ]
26  }
27 }
```

## **.dockerignore**

```
1 target/  
2 .git/  
3 .github/  
4 bench-baseline/  
5 *.pdf
```

**.github/workflows/ci.yml**

```

1 name: CI
2
3 on:
4   push:
5     branches: [main]
6   pull_request:
7     branches: [main]
8
9 jobs:
10  # — Lint (fast) —————
11  # Surfaces formatting and clippy failures in ~1 min, independently of tests.
12  lint:
13    runs-on: ubuntu-latest
14    steps:
15      - uses: actions/checkout@v4
16      - uses: dtolnay/rust-toolchain@stable
17      - uses: Swatinem/rust-cache@v2
18      - run: cargo fmt -- --check
19      - run: cargo clippy --all-targets -- -D warnings
20
21  # — Coverage —————
22  # Runs all tests (via cargo llvm-cov nextest) and fails if line coverage drops
23  # below 85%. Generates an HTML report uploaded as an artifact for the pages job.
24  test:
25    runs-on: ubuntu-latest
26    steps:
27      - uses: actions/checkout@v4
28      - uses: dtolnay/rust-toolchain@stable
29        with:
30          components: llvm-tools-preview
31      - uses: Swatinem/rust-cache@v2
32      - uses: taiki-e/install-action@v2
33        with:
34          tool: cargo-llvm-cov,nextest
35      - run: cargo llvm-cov nextest --lcov --output-path lcov.info --fail-under-lines 85
36      - run: cargo llvm-cov report --html
37      - uses: codecov/codecov-action@v5
38        with:
39          files: lcov.info
40          fail_ci_if_error: false
41      - uses: actions/upload-artifact@v4
42        with:
43          name: coverage-report
44          path: target/llvm-cov/html/
45          retention-days: 1
46      - name: Upload coverage reports to Codecov
47        uses: codecov/codecov-action@v5
48        with:
49          token: ${ secrets.CODECOV_TOKEN }
50
51  # — Binary —————
52  # Compiles a musl release binary in parallel with lint/test. Uses thin LTO
53  # (acceptable for nightly; fat LTO is reserved for shipped release binaries).
54  # The package and pages jobs download this artifact instead of recompiling.
55  build:
56    runs-on: ubuntu-latest
57    steps:
58      - uses: actions/checkout@v4
59      - uses: dtolnay/rust-toolchain@stable
60      - name: Install musl toolchain
61        run: sudo apt-get install -y musl-tools
62      - run: rustup target add x86_64-unknown-linux-musl
63      - uses: Swatinem/rust-cache@v2
64        with:
65          key: musl
66      - name: Build
67        env:
68          CARGO_PROFILE_RELEASE_LTO: thin
69          CARGO_PROFILE_RELEASE_CODEGEN_UNITS: "16"
70        run: cargo build --release --target x86_64-unknown-linux-musl
71      - run: tar czf gitprint-linux-musl.tar.gz -C target/x86_64-unknown-linux-musl/release gitprint
72      - uses: actions/upload-artifact@v4
73        with:
74          name: gitprint-linux-musl
75          path: gitprint-linux-musl.tar.gz
76          retention-days: 1
77

```

```
78 # — Docker image
79 # Downloads the pre-built musl binary from the build job and copies it into
80 # Alpine via --target prebuilt - no Rust compilation inside Docker.
81 #
82 # Push policy:
83 # - main branch → ghcr.io/.../gitprint:nightly + sha-<hash>
84 # - other branches / same-repo PRs → ghcr.io/.../gitprint:<branch> or pr-<n>
85 # - fork PRs → build only, no push (fork tokens lack GHCR write access)
86 package:
87   needs: [lint, test, build]
88   runs-on: ubuntu-latest
89   permissions:
90     packages: write
91   steps:
92     - uses: actions/checkout@v4
93
94     - uses: docker/setup-buildx-action@v3
95
96     # Login is skipped for fork PRs - their GITHUB_TOKEN cannot write to the
97     # upstream registry. Same-repo PRs and all push events have write access.
98     - uses: docker/login-action@v3
99       if: >-
100         github.event_name == 'push' ||
101         github.event.pull_request.head.repo.full_name == github.repository
102     with:
103       registry: ghcr.io
104       username: ${ github.actor }
105       password: ${ secrets.GITHUB_TOKEN }
106
107     # Tags:
108     # push to main → nightly + sha-<hash>
109     # push to branch → <branch-name> + sha-<hash>
110     # same-repo PR → pr-<number> + sha-<hash>
111     - uses: docker/metadata-action@v5
112       id: meta
113       with:
114         images: ghcr.io/${ github.repository }
115         tags: |
116           type=raw,value=nightly,enable=${ github.ref == 'refs/heads/main' }
117           type=ref,event=branch
118           type=ref,event=pr
119           type=sha,prefix=sha-,format=short
120
121     - uses: actions/download-artifact@v4
122       with:
123         name: gitprint-linux-musl
124         path: dist
125     - run: tar xzf dist/gitprint-linux-musl.tar.gz -C dist
126
127     - uses: docker/build-push-action@v6
128       with:
129         context: dist
130         file: Dockerfile
131         target: prebuilt
132         push: >-
133           ${
134             github.event_name == 'push' ||
135             github.event.pull_request.head.repo.full_name == github.repository
136           }
137         tags: ${ steps.meta.outputs.tags }
138         labels: ${ steps.meta.outputs.labels }
139
140 # — Pages (main only)
141 # Reuses the pre-built musl binary to generate the source PDF (skips
142 # cargo build --release), then builds rustdoc and deploys both to GitHub Pages.
143 # Runs only on main; concurrency guard cancels stale in-flight deployments.
144 pages:
145   if: github.ref == 'refs/heads/main'
146   needs: [lint, test, build]
147   runs-on: ubuntu-latest
148   concurrency:
149     group: pages
150     cancel-in-progress: true
151   permissions:
152     contents: read
153     pages: write
154     id-token: write
155   environment:
```

```
156     name: github-pages
157     url: ${ steps.deploy.outputs.page_url }}
158 steps:
159   - uses: actions/checkout@v4
160     with:
161       # Full history required so git log reports accurate per-file dates.
162       fetch-depth: 0
163
164   - uses: dtolnay/rust-toolchain@stable
165   - uses: Swatinem/rust-cache@v2
166
167   # Reuse the already-compiled musl binary – no cargo build --release needed.
168   - uses: actions/download-artifact@v4
169     with:
170       name: gitprint-linux-musl
171       path: dist
172   - run: tar xzf dist/gitprint-linux-musl.tar.gz -C dist && chmod +x dist/gitprint
173
174   - uses: actions/download-artifact@v4
175     with:
176       name: coverage-report
177       path: coverage-report
178
179   - name: Generate PDF
180     run: ./dist/gitprint . --output gitprint.pdf
181
182   - name: Generate rustdoc
183     run: cargo doc --no-deps
184
185   - name: Assemble site
186     env:
187       REPO: ${ github.repository }}
188     run: |
189       mkdir -p _site/docs _site/coverage
190       cp gitprint.pdf _site/gitprint.pdf
191       cp -r target/doc/. _site/docs/
192       cp -r coverage-report/. _site/coverage/
193       touch _site/.nojekyll
194       cat > _site/index.html << HTMLEOF
195       <!DOCTYPE html>
196       <html lang="en">
197       <head>
198         <meta charset="UTF-8">
199         <meta name="viewport" content="width=device-width, initial-scale=1.0">
200         <title>gitprint – source PDF</title>
201         <style>
202           * { margin: 0; padding: 0; box-sizing: border-box; }
203           html, body { height: 100%; font-family: system-ui, -apple-system, sans-serif; background: #0d1117; }
204           body { display: flex; flex-direction: column; }
205           header { flex-shrink: 0; display: flex; align-items: center; gap: 16px; padding: 10px 20px; background-color: #0d1117; }
206           header h1 { font-size: 15px; font-weight: 600; letter-spacing: -.01em; }
207           header a { font-size: 13px; color: #8b949e; text-decoration: none; }
208           header a:hover { color: #6edf33; }
209           embed { flex: 1; width: 100%; border: none; }
210         </style>
211       </head>
212       <body>
213         <header>
214           <h1>gitprint</h1>
215           <a href="https://github.com/${REPO}">source</a>
216           <a href="gitprint.pdf">download PDF</a>
217           <a href="docs/gitprint/index.html">API docs</a>
218           <a href="coverage/index.html">coverage</a>
219         </header>
220         <embed src="gitprint.pdf" type="application/pdf">
221       </body>
222     </html>
223     HTMLEOF
224
225   - uses: actions/configure-pages@v4
226   - uses: actions/upload-pages-artifact@v3
227     with:
228       path: _site
229   - id: deploy
230     uses: actions/deploy-pages@v4
231
232 # — Benchmarks (main only)
233 # Run all benchmarks and record results as "current". No regression
```

```
234 # comparison here - GitHub-hosted runners have too much performance
235 # variance between runs to produce a reliable baseline. The authoritative
236 # check runs locally via `make bench-check` / `make release`, where
237 # hardware is consistent. Skipped on PRs since results would be discarded.
238 benchmark:
239   if: github.ref == 'refs/heads/main'
240   runs-on: ubuntu-latest
241   steps:
242     - uses: actions/checkout@v4
243     - uses: dtolnay/rust-toolchain@stable
244     - uses: Swatinem/rust-cache@v2
245     - name: Run benchmarks
246       run: cargo bench --bench pipeline -- --save-baseline current
247     - uses: actions/upload-artifact@v4
248       with:
249         name: benchmark-report
250         path: target/criterion/
251         retention-days: 90
```

## .github/workflows/cleanup.yml

```
1 name: Cleanup GHCR
2
3 # Runs weekly and on demand. Deletes old container versions that are not
4 # protected by a semver tag, `latest`, or `nightly`. This covers ephemeral
5 # branch and PR images that accumulate over time.
6 #
7 # Protected tag regex: latest | nightly | X.Y.Z | X.Y
8 # Everything else (sha-*, branch names, pr-*) is subject to deletion once
9 # the number of unprotected versions exceeds min-versions-to-keep.
10
11 on:
12   schedule:
13     - cron: '0 3 * * 0' # every Sunday at 03:00 UTC
14   workflow_dispatch:
15
16 jobs:
17   cleanup:
18     runs-on: ubuntu-latest
19     permissions:
20       packages: write
21     steps:
22     - uses: actions/delete-package-versions@v5
23       with:
24         package-name: gitprint
25         package-type: container
26         token: ${ secrets.GITHUB_TOKEN }
27         # Keep at least the 25 most-recent unprotected versions so in-flight
28         # branch images stay available for a reasonable window before deletion.
29         min-versions-to-keep: 25
30         # Never delete semver releases, latest, or the nightly image.
31         ignore-versions: '^(latest|nightly|\d+\.\d+(\.\d+)?)$'
```

**.github/workflows/release.yml**

```

1 name: Release Binaries
2
3 # Builds native binaries for every major platform and publishes them on
4 # versioned tag pushes (e.g. v1.2.3). Nightly Docker images are published
5 # on every main-branch commit by ci.yml - no duplication needed here.
6
7 on:
8   push:
9     tags: ['v*']
10
11 env:
12   CARGO_TERM_COLOR: always
13
14 jobs:
15   # — Build —————
16   build:
17     name: Build · ${ matrix.target }
18     runs-on: ${ matrix.os }
19     strategy:
20       fail-fast: false
21     matrix:
22       include:
23         # Linux glibc - preferred by bininstall on mainstream distros; uses glibc's fast allocator
24         - target: x86_64-unknown-linux-gnu
25           os: ubuntu-latest
26           cross: false
27
28         # Linux musl - statically linked, runs on Alpine / musl-based distros
29         - target: x86_64-unknown-linux-musl
30           os: ubuntu-latest
31           cross: false
32
33         - target: aarch64-unknown-linux-musl
34           os: ubuntu-latest
35           cross: true # cross-compiled via the `cross` Docker toolchain
36
37         # macOS ARM (macos-14+ runner is ARM64)
38         - target: aarch64-apple-darwin
39           os: macos-latest
40           cross: false
41
42         # Windows
43         - target: x86_64-pc-windows-msvc
44           os: windows-latest
45           cross: false
46
47     steps:
48       - uses: actions/checkout@v4
49
50       # Toolchain version comes from rust-toolchain.toml; targets are per-matrix.
51       - uses: dtolnay/rust-toolchain@stable
52         with:
53           targets: ${ matrix.target }
54
55       # Belt-and-suspenders: ensure the target stdlib is installed even on a cold
56       # tool-cache (e.g. after a version bump in rust-toolchain.toml). cross builds
57       # use their own Docker image so only native builds need this.
58       - name: Ensure Rust target stdlib is installed
59         if: "!matrix.cross"
60         run: rustup target add ${ matrix.target }
61
62       - uses: Swatinem/rust-cache@v2
63         with:
64           key: release-${ matrix.target }
65
66       # musl-tools provides musl-gcc required for the x86_64-musl target.
67       - name: Install musl toolchain
68         if: contains(matrix.target, 'musl') && !matrix.cross
69         run: sudo apt-get install -y musl-tools
70
71       # taiki-e/install-action downloads pre-built cross binaries - much faster than
72       # compiling from source with `cargo install cross`.
73       - name: Install cross
74         if: matrix.cross
75         uses: taiki-e/install-action@v2
76         with:
77           tool: cross

```

```
78
79 - name: Build
80   shell: bash
81   env:
82     # Explicitly reinforce all profile.release optimizations from Cargo.toml so
83     # they can't be silently overridden by toolchain defaults or stale cache.
84     CARGO_PROFILE_RELEASE_LTO: fat
85     CARGO_PROFILE_RELEASE_CODEGEN_UNITS: "1"
86     CARGO_PROFILE_RELEASE_OPT_LEVEL: "3"
87     CARGO_PROFILE_RELEASE_PANIC: abort
88     CARGO_PROFILE_RELEASE_STRIP: symbols
89   run: |
90     if [[ "${{ matrix.cross }}" == "true" ]]; then
91       cross build --release --target ${{ matrix.target }}
92     else
93       cargo build --release --target ${{ matrix.target }}
94     fi
95
96 - name: Package (Unix)
97   if: runner.os != 'Windows'
98   run: >
99     tar czf gitprint-${{ matrix.target }}.tar.gz
100     -C target/${{ matrix.target }}/release gitprint
101
102 - name: Package (Windows)
103   if: runner.os == 'Windows'
104   shell: pwsh
105   run: >
106     Compress-Archive
107     -Path target/${{ matrix.target }}/release/gitprint.exe
108     -DestinationPath gitprint-${{ matrix.target }}.zip
109
110 - uses: actions/upload-artifact@v4
111   with:
112     name: gitprint-${{ matrix.target }}
113     # Each job uploads exactly one archive; glob picks whichever extension was created.
114     path: gitprint-${{ matrix.target }}.*
115
116 # — Docker image —————
117 # Publishes ghcr.io/izelnakri/gitprint:<version>, <major>.<minor>, and latest.
118 package:
119   name: Publish Docker image
120   needs: [build]
121   runs-on: ubuntu-latest
122   permissions:
123     packages: write
124   steps:
125     - uses: actions/checkout@v4
126
127     - uses: docker/setup-buildx-action@v3
128
129     - uses: docker/login-action@v3
130     with:
131       registry: ghcr.io
132       username: ${{ github.actor }}
133       password: ${{ secrets.GITHUB_TOKEN }}
134
135     # Derives: v0.3.4 → tags 0.3.4, 0.3, and latest.
136     - uses: docker/metadata-action@v5
137       id: meta
138     with:
139       images: ghcr.io/${{ github.repository }}
140       tags: |
141         type=semver,pattern={{version}}
142         type=semver,pattern={{major}}.{{minor}}
143         type=raw,value=latest
144
145     # Download the already-compiled musl binary from the build job – no recompilation needed.
146     - uses: actions/download-artifact@v4
147     with:
148       name: gitprint-x86_64-unknown-linux-musl
149       path: dist
150     - run: tar xzf dist/gitprint-x86_64-unknown-linux-musl.tar.gz -C dist
151
152     # Target the prebuilt stage: copies the binary into Alpine, skips all compile stages.
153     - uses: docker/build-push-action@v6
154     with:
155       context: dist
```

```
156     file: Dockerfile
157     target: prebuilt
158     push: true
159     tags: ${ steps.meta.outputs.tags }
160     labels: ${ steps.meta.outputs.labels }
161
162 # — Publish —————
163 publish:
164   name: Publish release
165   needs: [build]
166   runs-on: ubuntu-latest
167   permissions:
168     contents: write # required to create/delete GitHub releases and tags
169
170   steps:
171     - uses: actions/checkout@v4
172       with:
173         fetch-depth: 1
174
175     # Download all platform archives into a single directory.
176     - uses: actions/download-artifact@v4
177       with:
178         merge-multiple: true
179         path: artifacts
180
181     - name: Generate SHA-256 checksums
182       run: cd artifacts && sha256sum * > checksums.txt
183
184     # make release creates the GitHub Release entry locally (with CHANGELOG notes)
185     # before pushing the tag, so CI only needs to upload the built artifacts.
186     # The create fallback handles the rare case where CI wins the race.
187     - name: Upload artifacts to versioned release
188       env:
189         GH_TOKEN: ${ github.token }
190         GH_REPO: ${ github.repository }
191       run: |
192         gh release create "${ github.ref_name }" \
193           --title "${ github.ref_name }" \
194           --generate-notes 2>/dev/null || true
195         gh release upload "${ github.ref_name }" --clobber artifacts/*
```

## **.gitignore**

```
1 /target
2 /result
3 /bench-baseline
4 *.pdf
5 TODO
```

CHANGELOG.md

```

1 # ChangeLog
2
3 ## [0.4.0] - 2026-03-04
4 [ `v0.3.23...v0.4.0` ](https://github.com/izeInakri/gitprint/compare/v0.3.23...v0.4.0)
5
6 ### Features
7 - --preview mode!! - 2026-03-04 by [@izeInakri] (82eb371`) (https://github.com/izeInakri/commit/82eb371)
8
9 ## [0.3.23] - 2026-03-04
10 [ `v0.3.21...v0.3.23` ](https://github.com/izeInakri/gitprint/compare/v0.3.21...v0.3.23)
11
12 ### Bug Fixes
13 - V0.3.22 already exists - 2026-03-04 by [@izeInakri] (0184026`) (https://github.com/izeInakri/commit/0184026)
14
15 ## [0.3.21] - 2026-03-04
16 [ `v0.3.20...v0.3.21` ](https://github.com/izeInakri/gitprint/compare/v0.3.20...v0.3.21)
17
18 ### Performance
19 - Improve CI by caching musl for cargo release - 2026-03-04 by [@izeInakri] (b4a9`) (https://github.com/izeInakri/commit/b4a9)
20
21 ## [0.3.20] - 2026-03-03
22 [ `v0.3.19...v0.3.20` ](https://github.com/izeInakri/gitprint/compare/v0.3.19...v0.3.20)
23
24 ### Performance
25 - Parallelise CI, add nextest, skip nightly binary builds on main push - 2026-03-03 by [@izeInakri] (3111111`) (https://github.com/izeInakri/commit/3111111)
26
27 ## [0.3.18] - 2026-03-03
28 [ `v0.3.17...v0.3.18` ](https://github.com/izeInakri/gitprint/compare/v0.3.17...v0.3.18)
29
30 ### Performance
31 - Override fat LTO in Docker builds to cut link time from 3+ min to ~20 s - 2026-03-03 by [@izeInakri] (1111111`) (https://github.com/izeInakri/commit/1111111)
32
33 ## [0.3.17] - 2026-03-03
34 [ `v0.3.16...v0.3.17` ](https://github.com/izeInakri/gitprint/compare/v0.3.16...v0.3.17)
35
36 ### Bug Fixes
37 - Use pre-built lukemathwalker/cargo-chef image instead of cargo install - 2026-03-03 by [@izeInakri] (2222222`) (https://github.com/izeInakri/commit/2222222)
38
39 ### Performance
40 - Use cargo-chef + pinned Rust in Dockerfile for reliable layer caching - 2026-03-03 by [@izeInakri] (3333333`) (https://github.com/izeInakri/commit/3333333)
41
42 ## [0.3.16] - 2026-03-03
43 [ `v0.3.15...v0.3.16` ](https://github.com/izeInakri/gitprint/compare/v0.3.15...v0.3.16)
44
45 ### Bug Fixes
46 - Add stub benches/pipeline.rs to Dockerfile deps layer - 2026-03-03 by [@izeInakri] (4444444`) (https://github.com/izeInakri/commit/4444444)
47
48 ### Performance
49 - Replace build-musl + pre-built binary with multi-stage Dockerfile - 2026-03-03 by [@izeInakri] (5555555`) (https://github.com/izeInakri/commit/5555555)
50
51 ## [0.3.15] - 2026-03-03
52 [ `v0.3.14...v0.3.15` ](https://github.com/izeInakri/gitprint/compare/v0.3.14...v0.3.15)
53
54 ### Bug Fixes
55 - Drop rust:alpine container for build-musl, use musl-tools instead - 2026-03-03 by [@izeInakri] (6666666`) (https://github.com/izeInakri/commit/6666666)
56
57 ## [0.3.14] - 2026-03-03
58 [ `v0.3.13...v0.3.14` ](https://github.com/izeInakri/gitprint/compare/v0.3.13...v0.3.14)
59
60 ### Bug Fixes
61 - Download musl binary before Docker build in release package job - 2026-03-03 by [@izeInakri] (7777777`) (https://github.com/izeInakri/commit/7777777)
62
63 ## [0.3.13] - 2026-03-03
64 [ `v0.3.12...v0.3.13` ](https://github.com/izeInakri/gitprint/compare/v0.3.12...v0.3.13)
65
66 ### Bug Fixes
67 - Remove benchmark regression comparison from CI - 2026-03-02 by [@izeInakri] (8888888`) (https://github.com/izeInakri/commit/8888888)
68 - Remove benchmark regression check from release.yml - 2026-03-02 by [@izeInakri] (9999999`) (https://github.com/izeInakri/commit/9999999)
69
70 ## [0.3.12] - 2026-03-02
71 [ `v0.3.11...v0.3.12` ](https://github.com/izeInakri/gitprint/compare/v0.3.11...v0.3.12)
72
73 ### Bug Fixes
74 - Mark benchmark job continue-on-error in CI - 2026-03-02 by [@izeInakri] (8ad6a6`) (https://github.com/izeInakri/commit/8ad6a6)
75
76 ## [0.3.11] - 2026-03-02
77 [ `v0.3.10...v0.3.11` ](https://github.com/izeInakri/gitprint/compare/v0.3.10...v0.3.11)

```

```
78
79 ### Bug Fixes
80 - Use actions/cache with explicit paths for musl build caching - 2026-03-02 by [@izeInakri](https://github.com/izeInakri) ([`b5f5dd8`](https://github.com/izeInakri/commit/b5f5dd8))
81 - Fetch tags from remote before --list-tags on shallow clones - 2026-03-02 by [@izeInakri](https://github.com/izeInakri) ([`b5f5dd8`](https://github.com/izeInakri/commit/b5f5dd8))
82
83 ### Performance
84 - Improve "build-musl" CI workflow time - 2026-03-02 by [@izeInakri](https://github.com/izeInakri) ([`b5f5dd8`](https://github.com/izeInakri/commit/b5f5dd8))
85
86 ## [0.3.9] - 2026-03-02
87 [v0.3.8...v0.3.9](https://github.com/izeInakri/gitprint/compare/v0.3.8...v0.3.9)
88
89 ### Performance
90 - Run benchmarks in background during changelog preview in make release - 2026-03-02 by [@izeInakri](https://github.com/izeInakri) ([`b5f5dd8`](https://github.com/izeInakri/commit/b5f5dd8))
91
92 ## [0.3.8] - 2026-03-02
93 [v0.3.7...v0.3.8](https://github.com/izeInakri/gitprint/compare/v0.3.7...v0.3.8)
94
95 ### Bug Fixes
96 - Wire musl-gcc as linker and C compiler for musl CI build - 2026-03-02 by [@izeInakri](https://github.com/izeInakri) ([`b5f5dd8`](https://github.com/izeInakri/commit/b5f5dd8))
97 - Build musl binary natively in rust:alpine container - 2026-03-02 by [@izeInakri](https://github.com/izeInakri) ([`b5f5dd8`](https://github.com/izeInakri/commit/b5f5dd8))
98
99 ### Features
100 - --last-repos, --list-tags, date aliases, offline mock tests - 2026-03-02 by [@izeInakri](https://github.com/izeInakri) ([`b5f5dd8`](https://github.com/izeInakri/commit/b5f5dd8))
101 - Show version and binary size on one line in --help footer - 2026-03-02 by [@izeInakri](https://github.com/izeInakri) ([`b5f5dd8`](https://github.com/izeInakri/commit/b5f5dd8))
102 - Show git branch as version in cargo run; fix musl CI build - 2026-03-02 by [@izeInakri](https://github.com/izeInakri) ([`b5f5dd8`](https://github.com/izeInakri/commit/b5f5dd8))
103
104 ### Performance
105 - Pre-build musl binary in CI; Docker just copies it - 2026-03-02 by [@izeInakri](https://github.com/izeInakri) ([`b5f5dd8`](https://github.com/izeInakri/commit/b5f5dd8))
106
107 ## [0.3.7] - 2026-03-01
108 [v0.3.6...v0.3.7](https://github.com/izeInakri/gitprint/compare/v0.3.6...v0.3.7)
109
110 ### Features
111 - Now users can do $ gitprint -u izeInakri --commits 100 - 2026-02-20 by [@izeInakri](https://github.com/izeInakri) ([`a849ab9`](https://github.com/izeInakri/commit/a849ab9))
112 - Neon diff palette, rate-limit errors & user report refactor - 2026-03-01 by [@izeInakri](https://github.com/izeInakri) ([`a849ab9`](https://github.com/izeInakri/commit/a849ab9))
113
114 ## [0.3.6] - 2026-02-20
115 [v0.3.5...v0.3.6](https://github.com/izeInakri/gitprint/compare/v0.3.5...v0.3.6)
116
117 ### Documentation
118 - Add Docker installation guide & nix-aware make release - 2026-02-19 by [@izeInakri](https://github.com/izeInakri) ([`a849ab9`](https://github.com/izeInakri/commit/a849ab9))
119
120 ### Features
121 - Fix FS Size display & add docker nightly builds - 2026-02-19 by [@izeInakri](https://github.com/izeInakri) ([`a849ab9`](https://github.com/izeInakri/commit/a849ab9))
122 - -u flag now generates user reports! - 2026-02-20 by [@izeInakri](https://github.com/izeInakri) ([`a849ab9`](https://github.com/izeInakri/commit/a849ab9))
123
124 ## [0.3.5] - 2026-02-19
125 [v0.3.4...v0.3.5](https://github.com/izeInakri/gitprint/compare/v0.3.4...v0.3.5)
126
127 ### Features
128 - Metadata Page "Size" is now "Repo Size" or "FS Size" - 2026-02-19 by [@izeInakri](https://github.com/izeInakri) ([`e564a7e`](https://github.com/izeInakri/commit/e564a7e))
129 - ONLY ALLOW RELEASE OF BINARY mean benchmark < $REGRESSION_TRESHOLD - 2026-02-19 by [@izeInakri](https://github.com/izeInakri) ([`e564a7e`](https://github.com/izeInakri/commit/e564a7e))
130
131 ## [0.3.3] - 2026-02-19
132 [v0.3.2...v0.3.3](https://github.com/izeInakri/gitprint/compare/v0.3.2...v0.3.3)
133
134 ### Features
135 - Improve release workflow - 2026-02-18 by [@izeInakri](https://github.com/izeInakri) ([`e564a7e`](https://github.com/izeInakri/commit/e564a7e))
136 - Optimized release binaries, interactive release flow, CHANGELOG commit links - 2026-02-18 by [@izeInakri](https://github.com/izeInakri) ([`e564a7e`](https://github.com/izeInakri/commit/e564a7e))
137
138 ## [0.3.2] - 2026-02-18
139 [v0.3.1...v0.3.2](https://github.com/izeInakri/gitprint/compare/v0.3.1...v0.3.2)
140
141 ### Features
142 - Add make fix, cargo binstall support, and faster nix run - 2026-02-18 by [@izeInakri](https://github.com/izeInakri) ([`e564a7e`](https://github.com/izeInakri/commit/e564a7e))
143 - Accept remote URLs as input (git clone + generate PDF) - 2026-02-18 by [@izeInakri](https://github.com/izeInakri) ([`e564a7e`](https://github.com/izeInakri/commit/e564a7e))
144 - Improve Metadata first page - 2026-02-18 by [@izeInakri](https://github.com/izeInakri) ([`c7fc8d7`](https://github.com/izeInakri/commit/c7fc8d7))
145
146 ## [0.3.1] - 2026-02-18
147 [v0.3.0...v0.3.1](https://github.com/izeInakri/gitprint/compare/v0.3.0...v0.3.1)
148
149 ### Features
150 - Slim down dependency tree further - 2026-02-18 by [@izeInakri](https://github.com/izeInakri) ([`88ccf92`](https://github.com/izeInakri/commit/88ccf92))
151
152 ## [0.2.0] - 2026-02-18
153 [v0.1.2...v0.2.0](https://github.com/izeInakri/gitprint/compare/v0.1.2...v0.2.0)
154
155 ### Bug Fixes
```

```
156 - Ensure musl target stdlib is installed on cold cache; add README badges - 2026-02-18 by [izeInakri](https://gi
157
158 ### Features
159 - Add Makefile, release tooling, async save_pdf, and crates.io metadata - 2026-02-18 by [izeInakri](https://gith
160
161 ### Refactoring
162 - Migrate to anyhow, update all deps, parallelise pipeline, add doctests - 2026-02-18 by [izeInakri](https://git
163
164 ## [0.1.2] - 2026-02-18
165
166 ### Features
167 - Now gitprint shows more metadata of files - 2026-02-18 by [izeInakri](https://github.com/izeInakri) ([`514f0cb
168
169
```

```
1 # gitprint
2
3 Rust CLI that converts git repositories into syntax-highlighted, printer-friendly PDFs.
4
5 ## Build & Test
6
7 - `make check` - fmt check + clippy + tests (run before every commit)
8 - `make build` - build the project
9 - `make test` - run all tests (unit + integration)
10 - `make fmt` - format source code
11 - `make doc` - build and open API docs
12 - `make release [LEVEL=patch|minor|major]` - generate CHANGELOG and publish
13 - `nix flake check` - run all CI checks (build, clippy, fmt, tests)
14 - `nix develop` - enter development shell (includes git-cliff, cargo-release)
15
16 ## Architecture
17
18 Single-binary CLI. Pipeline: git → filter → highlight → PDF.
19
20 Modules:
21 - `cli.rs` - Clap argument parsing
22 - `types.rs` - Shared data types (Config, RepoMetadata, PaperSize, etc.)
23 - `git.rs` - Git operations via `git` CLI subprocess
24 - `filter.rs` - Glob-based file filtering + binary/minified detection
25 - `defaults.rs` - Default exclude glob patterns
26 - `highlight.rs` - Syntax highlighting via syntect
27 - `pdf/` - PDF generation via printpdf
28 - `mod.rs` - Document creation and writing
29 - `fonts.rs` - Embedded JetBrains Mono font loading
30 - `cover.rs` - Cover page rendering
31 - `toc.rs` - Table of contents rendering
32 - `tree.rs` - Directory tree visualization
33 - `code.rs` - Highlighted source code rendering
34 - `lib.rs` - Main pipeline orchestration
35 - `main.rs` - CLI entry point
36
37 ## Conventions
38
39 - Edition 2024
40 - Error handling: anyhow for ergonomic error propagation throughout. Always try to use `?` instead of `.unwrap()`,
41 it is possible.
42 - Have flat `if` checks if performance characteristics are same between 2 possible control flows.
43 - Tests: inline `#[cfg(test)] mod tests` for unit tests, `tests/` directory for integration tests. Make sure each
44 is well covered with tests.
45 - Integration tests use `tempfile` crate to create temporary git repos
46 - No unsafe code, if not needed.
47 - Code has to be always readable. Priorities are: Performance > Scalability > Readability > Maintainance
```

**Cargo.toml**

```

1 [package]
2 name = "gitprint"
3 version = "0.4.0"
4 edition = "2024"
5 description = "Convert git repositories into beautifully formatted, printer-friendly PDFs"
6 license = "MIT"
7 repository = "https://github.com/izelnakri/gitprint"
8 homepage = "https://github.com/izelnakri/gitprint"
9 readme = "README.md"
10 keywords = ["pdf", "git", "syntax-highlighting", "printing", "code"]
11 categories = ["command-line-utilities", "development-tools"]
12
13 [dependencies]
14 anyhow = "1"
15 clap = { version = "4", features = ["derive"] }
16 globset = "0.4"
17 printpdf = "0.9"
18 reqwest = { version = "0.12", default-features = false, features = ["json", "rustls-tls"] }
19 serde = { version = "1", features = ["derive"] }
20 serde_json = "1"
21 tokio = { version = "1", features = ["rt-multi-thread", "macros", "fs", "process"] }
22 syntect = { version = "5", default-features = false, features = ["default-syntaxes", "default-themes", "regex-fancy"] }
23
24 [dev-dependencies]
25 criterion = { version = "0.8", features = ["html_reports"] }
26 httpmock = "0.7"
27 tempfile = "3"
28 tokio = { version = "1", features = ["rt-multi-thread", "macros", "fs", "process"] }
29
30 [[bench]]
31 name = "pipeline"
32 harness = false
33
34 [package.metadata.release]
35 # Run git-cliff inside cargo-release's own transaction so CHANGELOG.md is
36 # written after the dirty check passes and included in the release commit.
37 pre-release-hook = ["git-cliff", "--tag", "v{{version}}", "-o", "CHANGELOG.md"]
38 # make release already runs `make check` (fmt + clippy + tests) immediately
39 # before invoking cargo-release, so skipping the duplicate test run here
40 # saves ~1-2 min without reducing safety.
41 verify = false
42
43 [package.metadata.binstall]
44 pkg-url = "{ repo }/releases/download/v{ version }/{ name }-{ target }-{ archive-suffix }"
45 bin-dir = "{ bin }-{ binary-ext }"
46
47 [profile.release]
48 opt-level = 3
49 lto = "fat"
50 codegen-units = 1
51 strip = true
52 panic = "abort"

```

## Dockerfile

```
1 # cargo-chef is pre-installed in this image – no source compilation needed.
2 # 0.1.75-rust-alpine3.23 is an immutable tag (pinned cargo-chef + Alpine version)
3 # so the chef and deps layers are never invalidated by a base image update.
4 # Bump the tag deliberately when upgrading Rust or cargo-chef.
5 FROM lukemathwalker/cargo-chef:0.1.75-rust-alpine3.23 AS chef
6 WORKDIR /app
7
8 # Planner: fast – inspects Cargo.toml/Cargo.lock and emits recipe.json.
9 FROM chef AS planner
10 COPY . .
11 RUN cargo chef prepare --recipe-path recipe.json
12
13 # Builder stage 1 – dependencies only.
14 # Cached until recipe.json changes (i.e. Cargo.lock / Cargo.toml change).
15 # LTO and codegen-units are overridden here: fat LTO + CU=1 cannot be
16 # layer-cached and would add 3+ min of link time on every push.
17 FROM chef AS builder
18 COPY --from=planner /app/recipe.json recipe.json
19 RUN CARGO_PROFILE_RELEASE_LTO=thin \
20     CARGO_PROFILE_RELEASE_CODEGEN_UNITS=16 \
21     cargo chef cook --release --recipe-path recipe.json
22
23 # Builder stage 2 – gitprint source only (~20-30 s on every push).
24 COPY . .
25 RUN CARGO_PROFILE_RELEASE_LTO=thin \
26     CARGO_PROFILE_RELEASE_CODEGEN_UNITS=16 \
27     cargo build --release
28
29 # Pre-built stage: copies an already-compiled musl binary from the build context.
30 # Used by both ci.yml and release.yml (--target prebuilt) to skip all compilation.
31 # Local full build: `docker build .` uses the default stage below instead.
32 FROM alpine:latest AS prebuilt
33 RUN apk add --no-cache git
34 COPY gitprint /usr/local/bin/gitprint
35 ENTRYPOINT ["gitprint"]
36
37 # Default stage: compiled from source via the builder above (local docker build).
38 FROM alpine:latest
39 RUN apk add --no-cache git
40 COPY --from=builder /app/target/release/gitprint /usr/local/bin/gitprint
41 ENTRYPOINT ["gitprint"]
```

## Makefile

```

1  .DEFAULT_GOAL := help
2
3  LEVEL ?= patch
4  REGRESSION_THRESHOLD ?= 20
5
6  .PHONY: help fix check fmt test build doc bench-baseline bench-check release coverage
7
8  help:
9  ① @echo "Usage: make <target> [LEVEL=patch|minor|major] [REGRESSION_THRESHOLD=20]"
10 ① @echo ""
11 ① @echo "  fix           Auto-fix formatting and clippy lints"
12 ① @echo "  check        Fmt check + clippy + tests (validation)"
13 ① @echo "  fmt          Format source code"
14 ① @echo "  test         Run all tests"
15 ① @echo "  build        Build the project"
16 ① @echo "  doc          Build and open API documentation"
17 ① @echo "  bench-baseline Establish (or reset) the local benchmark baseline"
18 ① @echo "  bench-check  Run benchmarks and report regressions vs baseline"
19 ① @echo "  coverage     Run tests with coverage and fail if line coverage < 85%"
20 ① @echo "  release     Preview changelog, confirm, then generate CHANGELOG and publish (LEVEL=patch)"
21
22 fix:
23 ① cargo fmt
24 ① cargo clippy --fix --allow-dirty --allow-staged --all-targets
25
26 check:
27 ① cargo fmt -- --check
28 ① cargo clippy --all-targets -- -D warnings
29 ① @cargo nextest --version >/dev/null 2>&1 && cargo nextest run || cargo test
30
31 fmt:
32 ① cargo fmt
33
34 test:
35 ① @cargo nextest --version >/dev/null 2>&1 && cargo nextest run || cargo test
36
37 build:
38 ① cargo build
39
40 doc:
41 ① cargo doc --no-deps --open
42
43 # Run all benchmarks and save results as the new "main" baseline.
44 # Run this once after cloning, or whenever you want to reset the reference point.
45 bench-baseline:
46 ① @echo "=== Establishing benchmark baseline ==="
47 ① cargo bench --bench pipeline -- --save-baseline current
48 ① REGRESSION_THRESHOLD=$(REGRESSION_THRESHOLD) python3 scripts/check_benchmarks.py --save
49 ① @echo "Done. Future 'make bench-check' and 'make release' will compare against this baseline."
50
51 # Run benchmarks as "current" and compare against the stored baseline.
52 # Exits non-zero if any benchmark regressed more than REGRESSION_THRESHOLD percent.
53 bench-check:
54 ① @echo "=== Benchmark regression check (threshold: $(REGRESSION_THRESHOLD)%) ==="
55 ① cargo bench --bench pipeline -- --save-baseline current
56 ① REGRESSION_THRESHOLD=$(REGRESSION_THRESHOLD) python3 scripts/check_benchmarks.py
57
58 coverage:
59 ① @cargo nextest --version >/dev/null 2>&1 && cargo llvm-cov nextest --html --open --fail-under-lines 85 || cargo
60
61 # Benchmarks run in the background while the developer reads the changelog
62 # preview - reading time is free CPU time. Results are checked after the
63 # developer confirms, before cargo-release runs. Aborts kill the background
64 # process cleanly. `fix` is intentionally excluded: auto-modifying code at
65 # release time is unsafe; run `make fix` manually if check fails.
66 # If not already inside a nix shell, re-enters via `nix develop` so git-cliff
67 # and cargo-release are available. `exec` replaces the shell process, preventing
68 # any subsequent commands from running outside nix.
69 release:
70 ① @if [ -z "$$IN_NIX_SHELL" ]; then \
71 ① ① echo "==> Entering nix develop (git-cliff, cargo-release)..."; \
72 ① ① exec nix develop --command $(MAKE) release LEVEL=$(LEVEL) REGRESSION_THRESHOLD=$(REGRESSION_THRESHOLD); \
73 ① fi; \
74 ① $(MAKE) check; \
75 ① cargo bench --bench pipeline -- --save-baseline current > /tmp/gitprint-bench.log 2>&1 & BENCH_PID=$$!; \
76 ① printf "\n=== Release Preview (level: $(LEVEL)) ===\n\n"; \
77 ① git-cliff --bump --unreleased 2>/dev/null || true; \

```

```
78 0 printf "\nProceed with $(LEVEL) release? [y/N] " > /dev/tty; \  
79 0 read confirm < /dev/tty; \  
80 0 case "$confirm" in \  
81 0  [yY]*) \  
82 0 0 printf "\n=== Benchmark results ===\n"; \  
83 0 0 tail -f /tmp/gitprint-bench.log & TAIL_PID=$$!; \  
84 0 0 wait $$BENCH_PID; BENCH_EXIT=$$?; \  
85 0 0 kill $$TAIL_PID 2>/dev/null; wait $$TAIL_PID 2>/dev/null; \  
86 0 0 [ $$BENCH_EXIT -eq 0 ] || { printf "\nBenchmark run failed.\n"; rm -f /tmp/gitprint-bench.log; exit 1; }; \  
87 0 0 REGRESSION_THRESHOLD=$(REGRESSION_THRESHOLD) python3 scripts/check_benchmarks.py || { rm -f /tmp/gitprint-bench.log; exit 1; }; \  
88 0 0 cargo release $(LEVEL) --execute; \  
89 0 0 TAG=$(git describe --tags --abbrev=0); \  
90 0 0 awk '/^## \[/^{if(found) exit; found=1} found' CHANGELOG.md > /tmp/release-notes.md; \  
91 0 0 gh release create "$TAG" --title "$TAG" --notes-file /tmp/release-notes.md; \  
92 0 0 rm -f /tmp/release-notes.md /tmp/gitprint-bench.log; \  
93 0 0 REGRESSION_THRESHOLD=$(REGRESSION_THRESHOLD) python3 scripts/check_benchmarks.py --save \  
94 0 0 ;; \  
95 0 *) printf "Aborted.\n"; kill $$BENCH_PID 2>/dev/null; rm -f /tmp/gitprint-bench.log; exit 1 ;; \  
96 0 esac \  
97 \  
98 demo: \  
99 0 vhs demo/demo.tape \  
100
```

```

1 <div align="center">
2
3 # gitprint
4
5 [[!CI]](https://github.com/izelnakri/gitprint/actions/workflows/ci.yml/badge.svg)](https://github.com/izelnakri/gi
6 [[!Coverage]](https://codecov.io/gh/izelnakri/gitprint/graph/badge.svg)](https://codecov.io/gh/izelnakri/gitprint)
7 [[!Crate]](https://img.shields.io/crates/v/gitprint)](https://crates.io/crates/gitprint)
8 [[!Downloads]](https://img.shields.io/crates/d/gitprint)](https://crates.io/crates/gitprint)
9 [[!Docs]](https://img.shields.io/badge/docs-online-blue)](https://izelnakri.github.io/gitprint/docs/gitprint/)
10 [[!Sponsor]](https://img.shields.io/badge/sponsor-%E2%99%A5-pink)](https://github.com/sponsors/izelnakri)
11
12 </div>
13
14 Convert git repositories into beautifully formatted, printer-friendly PDFs – or preview them directly in the term
15
16 ![gitprint demo](demo/demo.gif)
17
18 ## Features
19
20 - Syntax-highlighted source code with 100+ languages supported
21 - Configurable color themes (InspiredGitHub, Solarized, base16, and more)
22 - Table of contents and directory tree visualization
23 - Single-file mode – print just one file, no cover page or TOC overhead
24 - Plain directory support – works on any folder, not just git repos
25 - Automatic binary and minified file detection and exclusion
26 - Glob-based include/exclude filtering
27 - Multiple paper sizes (A4, Letter, Legal) and landscape mode
28 - Branch and commit selection for printing specific revisions
29 - Embedded JetBrains Mono font for crisp code rendering
30 - Async pipeline – metadata, file reads, and highlighting run concurrently
31 - **Terminal preview mode** – inspect repo or user data in the terminal without generating a PDF
32 - **GitHub user report mode** – generate a PDF (or preview) of a user's activity, repos, and recent commits
33
34 ## Installation
35
36 ### cargo-binstall (recommended – downloads pre-built binary)
37
38 ```sh
39 cargo binstall gitprint
40 ```
41
42 ### Pre-built binary
43
44 Download the latest release for your platform from the
45 [releases page](https://github.com/izelnakri/gitprint/releases).
46
47 ### Nix
48
49 ```sh
50 nix profile install github:izelnakri/gitprint
51 ```
52
53 ### Build from source
54
55 ```sh
56 cargo install --git https://github.com/izelnakri/gitprint
57 ```
58
59 ### With Docker
60
61 No install needed – pull the latest nightly image from GitHub Container Registry and mount your repository:
62
63 ```sh
64 docker run --rm -w /repo -v "$(pwd):/repo" ghcr.io/izelnakri/gitprint:nightly . -o output.pdf
65 ```
66
67 - `--rm` removes the container after use
68 - `-w /repo` sets the working directory inside the container so `.` resolves correctly
69 - `-v "$(pwd):/repo"` mounts the current directory; the output PDF is written back to it
70
71 ## Usage
72
73 ### Repository Mode (Default)
74
75 ```sh
76 # Generate PDF from current directory (or git repo)
77 gitprint .

```

```
78
79 # Print a single file
80 gitprint src/main.rs
81
82 # Print any directory (no git required)
83 gitprint /path/to/dir
84
85 # Print a remote repository
86 gitprint https://github.com/user/repo
87
88 # Output to a specific file
89 gitprint . -o output.pdf
90
91 # Include only Rust and TOML files
92 gitprint . --include "*.rs" --include "*.toml"
93
94 # Exclude test files
95 gitprint . --exclude "test_*.rs"
96
97 # Use a different theme
98 gitprint . --theme "Solarized (dark)"
99
100 # List available themes
101 gitprint . --list-themes
102
103 # Use Letter paper in landscape
104 gitprint . --paper-size letter --landscape
105
106 # Print a specific branch or commit
107 gitprint . --branch feature-x
108 gitprint . --commit abc1234
109
110 # Minimal output: no TOC, no file tree, no line numbers
111 gitprint . --no-toc --no-file-tree --no-line-numbers
112 ```
113
114 ### User Report Mode
115
116 ```sh
117 # Generate a GitHub user activity report PDF
118 gitprint --user torvalds
119
120 # Limit the activity date range
121 gitprint --user torvalds --since "last month"
122 gitprint --user torvalds --since 2024-01-01 --until 2024-12-31
123
124 # Show only push events (commits), not issues/PRs/stars
125 gitprint --user torvalds --activity commits
126
127 # Include more repos and commits in the report
128 gitprint --user torvalds --last-repos 10 --last-commits 10
129
130 # Skip commit diffs for a faster, lighter report
131 gitprint --user torvalds --no-diffs
132
133 # Increase GitHub API rate limits with a personal access token
134 GITHUB_TOKEN=ghp_... gitprint --user torvalds
135 ```
136
137 ### Preview Mode
138
139 Preview shows all the same data as the PDF – metadata, directory tree, file list with LOC/sizes, or GitHub user a
140
141 ```sh
142 # Preview a repository in the terminal
143 gitprint . --preview
144
145 # Preview a remote repository
146 gitprint https://github.com/user/repo --preview
147
148 # Preview a GitHub user report
149 gitprint --user torvalds --preview
150
151 # Combine with other flags – filters and date ranges all apply
152 gitprint . --preview --include "*.rs"
153 gitprint --user torvalds --preview --since "last month" --activity commits
154 ```
155
```

156 **## CLI Reference**

157  
158 `````

159 Convert git repositories into beautifully formatted PDFs.

160  
161 **MODES**

162  
163 gitprint <PATH> [OPTIONS]  
164 Local path, file, or remote URL (https://, git@, ssh://) → PDF

165  
166 gitprint --user <USERNAME> [OPTIONS]  
167 GitHub user activity report → PDF

168  
169 gitprint <PATH|--user USERNAME> --preview  
170 Preview output in the terminal – no PDF generated

171  
172 Usage: gitprint [OPTIONS] [PATH]

173  
174 Arguments:  
175 [PATH]

176 Local path, file, or remote URL (https://, git@, ssh://)

177  
178 Options:

179 --preview Preview output in the terminal instead of generating a PDF  
180 -o, --output <PATH> Output PDF file path  
181 -h, --help Print help  
182 -V, --version Print version

183  
184 Repository Mode (Default):

185 --include <PATTERN> Glob patterns for files to include (repeatable)  
186 --exclude <PATTERN> Glob patterns for files to exclude (repeatable)  
187 --theme <NAME> Syntax highlighting theme [default: InspiredGitHub]  
188 --font-size <SIZE> Code font size in points [default: 8]  
189 --no-line-numbers Disable line numbers  
190 --no-toc Disable table of contents  
191 --no-file-tree Disable directory tree visualization  
192 --branch <NAME> Use a specific branch  
193 --commit <HASH> Use a specific commit  
194 --paper-size <SIZE> Paper size [default: a4] [possible values: a4, letter, legal]  
195 --landscape Use landscape orientation  
196 --list-themes List available syntax themes and exit  
197 --list-tags List version tags of the repository and exit

198  
199 User Report Mode:

200 -u, --user <USERNAME> GitHub username – generate a user activity report  
201 --last-repos <N> Most-recently-pushed repos to include [default: 5]  
202 --last-commits <N> Recent commits with diffs to render [default: 5]  
203 --no-diffs Skip commit diff rendering (faster)  
204 --since <DATE> Show events from this date forward  
205 --until <DATE> Show events up to and including this date  
206 --activity <TYPE> Event types: all (default) or commits  
207 --events <N> Max events shown in activity feed [default: 30]

208 `````

209  
210 **### Date formats for `--since` / `--until`**

211  
212 | Format | Example |  
213 |-----|-----|  
214 | ISO date | `2024-01-15` or `2024-01-15T00:00:00` |  
215 | Keywords | `today`, `yesterday` |  
216 | Named | `last week`, `last month`, `last year` |  
217 | Relative | `30 days ago`, `2 weeks ago`, `1 month ago` |

218  
219 **## Development**

220  
221 `sh`  
222 nix develop # Enter dev shell (Rust toolchain, git-cliff, cargo-release)  
223 make check # Fmt check + clippy + tests (run before every commit)  
224 make build # Build  
225 make test # Run tests  
226 make doc # Build and open API docs  
227 make release # Bump CHANGELOG + publish (LEVEL=patch|minor|major)  
228 nix flake check # Full CI suite: build, clippy, fmt, tests  
229 ``````

230  
231 **## Donate**

232  
233 If gitprint saves you time, consider sponsoring development:

```
234
235 **[github.com/sponsors/izelnakri](https://github.com/sponsors/izelnakri)**
236
237 GitHub Sponsors has zero platform fees – 100% goes to the developer.
238
239 ## License
240
241 MIT
```

## benches/pipeline.rs

```

1 use std::path::Path;
2
3 use std::hint::black_box;
4
5 use criterion::{Criterion, criterion_group, criterion_main};
6
7 use gitprint::filter::FileFilter;
8 use gitprint::highlight::Highlighter;
9 use gitprint::types::HighlightedLine;
10
11 const SAMPLE_RUST: &str = r#"
12 use std::collections::HashMap;
13
14 fn main() {
15     let mut map = HashMap::new();
16     map.insert("key", 42);
17
18     for (k, v) in &map {
19         println!("{k}: {v}");
20     }
21
22     let result: Vec<_> = (0..100)
23         .filter(|n| n % 2 == 0)
24         .map(|n| n * n)
25         .collect();
26
27     println!("{result:?}");
28 }
29 "#;
30
31 fn bench_highlight(c: &mut Criterion) {
32     let highlighter = Highlighter::new("InspiredGitHub").unwrap();
33     let path = Path::new("sample.rs");
34
35     c.bench_function("highlight_rust_file", |b| {
36         b.iter(|| {
37             let lines: Vec<HighlightedLine> = highlighter
38                 .highlight_lines(black_box(SAMPLE_RUST), path)
39                 .collect();
40             black_box(lines);
41         });
42     });
43
44     let large_content = SAMPLE_RUST.repeat(50);
45     c.bench_function("highlight_large_file", |b| {
46         b.iter(|| {
47             let lines: Vec<HighlightedLine> = highlighter
48                 .highlight_lines(black_box(&large_content), path)
49                 .collect();
50             black_box(lines);
51         });
52     });
53 }
54
55 fn bench_filter(c: &mut Criterion) {
56     let paths: Vec<std::path::PathBuf> = (0..1000)
57         .flat_map(|i| {
58             vec![
59                 format!("src/module_{i}/mod.rs").into(),
60                 format!("src/module_{i}/test.rs").into(),
61                 format!("docs/page_{i}.md").into(),
62                 format!("assets/image_{i}.png").into(),
63                 format!("node_modules/pkg_{i}/index.js").into(),
64             ]
65         })
66         .collect();
67
68     c.bench_function("filter_5000_paths", |b| {
69         b.iter(|| {
70             let filter = FileFilter::new(&["*.rs"].to_string(), &["*test*"].to_string()).unwrap();
71             let filtered: Vec<_> = filter.filter_paths(black_box(paths.clone())).collect();
72             black_box(filtered);
73         });
74     });
75 }
76
77 fn bench_highlighter_creation(c: &mut Criterion) {

```

```
78     c.bench_function("highlighter_new", |b| {
79         b.iter(|| {
80             black_box(Highlighter::new("InspiredGitHub").unwrap());
81         });
82     });
83 }
84
85 criterion_group!(
86     benches,
87     bench_highlight,
88     bench_filter,
89     bench_highlighter_creation
90 );
91 criterion_main!(benches);
```

## build.rs

```
1 fn main() {
2     // Rerun whenever the checked-out branch changes.
3     println!("cargo:rerun-if-changed=.git/HEAD");
4
5     // Only embed the branch name in debug builds so `cargo run` shows the
6     // current branch while release binaries show the Cargo.toml version.
7     if std::env::var("PROFILE").as_deref() != Ok("debug") {
8         return;
9     }
10
11     let branch = std::process::Command::new("git")
12         .args(["rev-parse", "--abbrev-ref", "HEAD"])
13         .output()
14         .ok()
15         .filter(|o| o.status.success())
16         .and_then(|o| String::from_utf8(o.stdout).ok())
17         .map(|s| s.trim().to_string())
18         .filter(|s| !s.is_empty() && s != "HEAD"); // HEAD = detached, skip
19
20     if let Some(branch) = branch {
21         println!("cargo:rustc-env=GITPRINT_GIT_BRANCH={branch}");
22     }
23 }
```

## cliff.toml

```

1 [changelog]
2 header = "# Changelog\n\n"
3 body = ""
4 {% set host = "https://github.com" -%}
5 {% set repo_url = host ~ "/" ~ remote.github.owner ~ "/" ~ remote.github.repo -%}
6 {% if version -%}
7 ## [{{ version | trim_start_matches(pat="v") }}] - {{ timestamp | date(format="%Y-%m-%d") }}
8 {% if previous.version %}[`{{ previous.version }}`...`{{ version }}`]({{ repo_url }}/compare/{{ previous.version }}).
9 {% endif %}
10 {% else -%}
11 ## [unreleased]
12 {% endif -%}
13 {% for group, commits in commits | group_by(attribute="group") -%}
14 ### {{ group | upper_first }}
15 {% for commit in commits -%}
16 - {% if commit.breaking %}[**breaking**] {% endif %}{{ commit.message | upper_first }} - {{ commit.author.timestamp }}
17 {% endfor %}
18 {% endfor %}
19 ""
20 footer = ""
21 trim = true
22
23 [git]
24 conventional_commits = true
25 filter_unconventional = true
26 split_commits = false
27 commit_parsers = [
28   { message = "^feat",      group = "Features" },
29   { message = "^fix",       group = "Bug Fixes" },
30   { message = "^perf",      group = "Performance" },
31   { message = "^refactor",  group = "Refactoring" },
32   { message = "^docs?",     group = "Documentation" },
33   { message = "^chore",     skip = true },
34   { message = "^ci",        skip = true },
35   { message = "^test",      skip = true },
36   { message = "^build",     skip = true },
37 ]
38 filter_commits = true
39 tag_pattern = "v[0-9].*"
40 skip_tags = "nightly"
41 topo_order = false
42 sort_commits = "oldest"
43
44 # — Hosting configuration —————
45 # git-cliff reads `owner` and `repo` from this section and exposes them as
46 # `remote.github.owner` / `remote.github.repo` in the body template above.
47 # Both values are auto-detected from `git remote get-url origin` when omitted.
48 #
49 # To switch providers:
50 # 1. Rename the section heading below (e.g. [remote.gitlab])
51 # 2. Update `host` in the body template to match the provider base URL
52 # 3. If needed, adjust URL path segments in the template:
53 #     GitLab   /compare/ → /-/compare/   /commit/ → /-/commit/
54 #     Bitbucket /compare/ → /branches/compare/ /commit/ → /commits/
55 #
56 # Provider section names git-cliff understands:
57 # [remote.github] - github.com
58 # [remote.gitlab] - gitlab.com (or self-hosted GitLab)
59 # [remote.gitea] - gitea.io (or self-hosted Gitea)
60 # [remote.forgejo] - forgejo.org (or self-hosted Forgejo)
61 # [remote.bitbucket] - bitbucket.org
62 [remote.github]
63 owner = "izeInakri"
64 repo = "gitprint"

```

## demo/demo.tape

```
1 Output demo/demo.gif
2
3 Set Shell "bash"
4 Set FontSize 16
5 Set Width 1200
6 Set Height 700
7 Set Theme "Catppuccin Mocha"
8 Set Padding 20
9 Set BorderRadius 12
10 Set WindowBar Colorful
11 Set Framerate 30
12 Set TypingSpeed 40ms
13
14 Hide
15 Type "export PATH=$HOME/.cargo/bin:$PATH && cd /home/izelnakri/Github/gitprint && clear"
16 Enter
17 Sleep 1s
18 Show
19
20 Type "# Generate a PDF of this repository"
21 Sleep 500ms
22 Enter
23 Sleep 300ms
24 Type "gitprint . -o demo/gitprint.pdf"
25 Sleep 300ms
26 Enter
27 Sleep 5s
28
29 Type "xdg-open demo/gitprint.pdf"
30 Sleep 300ms
31 Enter
32 Sleep 2s
33
34 Type "# Generate a GitHub user activity report"
35 Sleep 500ms
36 Enter
37 Sleep 300ms
38 Type "gitprint --user izelnakri -o demo/izelnakri.pdf"
39 Sleep 300ms
40 Enter
41 Sleep 5s
42
43 Type "xdg-open demo/izelnakri.pdf"
44 Sleep 300ms
45 Enter
46 Sleep 5s
47
48 Type "# Preview the repo structure in the terminal"
49 Sleep 500ms
50 Enter
51 Sleep 300ms
52 Type "gitprint --preview ."
53 Sleep 300ms
54 Enter
55 Sleep 5s
56
```

**flake.nix**

```

1 {
2   description = "Convert git repositories into beautifully formatted, printer-friendly PDFs";
3
4   inputs = {
5     nixpkgs.url = "github:NixOS/nixpkgs/nixpkgs-unstable";
6     flake-utils.url = "github:numtide/flake-utils";
7     crane.url = "github:ipetkov/crane";
8     rust-overlay = {
9       url = "github:oxalica/rust-overlay";
10      inputs.nixpkgs.follows = "nixpkgs";
11    };
12  };
13
14  outputs = { self, nixpkgs, flake-utils, crane, rust-overlay }:
15    flake-utils.lib.eachDefaultSystem (system:
16      let
17        overlays = [ (import rust-overlay) ];
18        pkgs = import nixpkgs { inherit system overlays; };
19
20        # Pin to the exact version declared in rust-toolchain.toml – same as CI.
21        # Dev shell adds rust-src + rust-analyzer on top for IDE support.
22        rustToolchain = (pkgs.rust-bin.fromRustupToolchainFile ./rust-toolchain.toml).override {
23          extensions = [ "rust-src" "rust-analyzer" "llvm-tools-preview" ];
24        };
25
26        craneLib = (crane.mkLib pkgs).overrideToolchain rustToolchain;
27
28        # Source filtering: include Rust/Cargo files + embedded fonts
29        unfilteredRoot = ./.;
30        src = pkgs.lib.fileset.toSource {
31          root = unfilteredRoot;
32          fileset = pkgs.lib.fileset.unions [
33            (craneLib.fileset.commonCargoSources unfilteredRoot)
34            (pkgs.lib.fileset.fileFilter (file: file.hasExt "ttf") unfilteredRoot)
35          ];
36        };
37
38        commonArgs = {
39          inherit src;
40          nativeBuildInputs = [ pkgs.pkg-config pkgs.makeWrapper ];
41        };
42
43        # Build dependencies only – cached separately from source changes
44        cargoArtifacts = craneLib.buildDepsOnly commonArgs;
45
46        # Build-only package: no tests so `nix run` and `nix build` are fast.
47        # Tests live in checks.tests below and still run on `nix flake check`.
48        gitprint = craneLib.buildPackage (commonArgs // {
49          inherit cargoArtifacts;
50          doCheck = false;
51
52          postInstall = ''
53            wrapProgram $out/bin/gitprint \
54              --prefix PATH : ${pkgs.lib.makeBinPath [ pkgs.git ]}
55          '';
56
57          meta = with pkgs.lib; {
58            description = "Convert git repositories into beautifully formatted, printer-friendly PDFs";
59            homepage = "https://github.com/izelnakri/gitprint";
60            license = licenses.mit;
61            mainProgram = "gitprint";
62          };
63        });
64      in
65      {
66        packages = {
67          default = gitprint;
68          gitprint = gitprint;
69        };
70
71        apps.default = flake-utils.lib.mkApp {
72          drv = gitprint;
73        };
74
75        devShells.default = pkgs.mkShell {
76          packages = [
77            rustToolchain

```

```
78     pkgs.git
79     pkgs.cargo-watch
80     pkgs.cargo-edit
81     pkgs.git-cliff
82     pkgs.cargo-release
83     pkgs.cargo-nextest
84     pkgs.cargo-llvm-cov
85     pkgs.vhs
86 ];
87
88 RUST_SRC_PATH = "${rustToolchain}/lib/rustlib/src/rust/library";
89 };
90
91 checks = {
92     inherit gitprint;
93
94     tests = craneLib.cargoTest (commonArgs // {
95         inherit cargoArtifacts;
96         nativeCheckInputs = [ pkgs.git ];
97         preCheck = ''
98             export HOME=$(mktemp -d)
99             '';
100     });
101
102     clippy = craneLib.cargoClippy (commonArgs // {
103         inherit cargoArtifacts;
104         cargoClippyExtraArgs = "--all-targets -- -D warnings";
105     });
106
107     fmt = craneLib.cargoFmt {
108         inherit src;
109     };
110 };
111 }
112 ) // {
113     overlays.default = final: prev: {
114         gitprint = self.packages.${prev.stdenv.hostPlatform.system}.default;
115     };
116 };
117 }
```

## rust-toolchain.toml

3 LOC · 88 B · 2026-03-14

```
1 [toolchain]
2 channel = "1.88.0"
3 components = ["rustfmt", "clippy", "llvm-tools-preview"]
```

**scripts/check\_benchmarks.py**

```

1  #!/usr/bin/env python3
2  """Benchmark regression checker for gitprint.
3
4  Modes
5  -----
6  Default (no flags)
7      Compare target/criterion/**/current/estimates.json against the stored
8      baseline in bench-baseline/**/main/estimates.json. Exits 1 if any
9      benchmark regressed more than REGRESSION_THRESHOLD percent (default 20).
10
11  --save
12      Promote target/criterion/**/current/ results to bench-baseline/**/main/
13      without running any comparison. Call this after a passing check to lock
14      in the current results as the new baseline.
15
16  The REGRESSION_THRESHOLD environment variable overrides the default (20%).
17
18  Usage
19  -----
20  # Check for regressions (fails fast):
21  python3 scripts/check_benchmarks.py
22
23  # Save current results as the new baseline (after a passing release):
24  python3 scripts/check_benchmarks.py --save
25  """
26  import glob
27  import json
28  import os
29  import shutil
30  import sys
31
32  CRITERION_DIR = "target/criterion"
33  BASELINE_DIR = "bench-baseline"
34
35
36  def _load_mean(path: str) -> float:
37      with open(path) as f:
38          return json.load(f)["mean"]["point_estimate"]
39
40
41  def save() -> None:
42      """Copy target/criterion/**/current/estimates.json → bench-baseline/**/main/..."""
43      saved = 0
44      for src in glob.glob(f"{CRITERION_DIR}/**/current/estimates.json", recursive=True):
45          rel = os.path.relpath(src, CRITERION_DIR)
46          dest = os.path.join(BASELINE_DIR, rel.replace("/current/", "/main/"))
47          os.makedirs(os.path.dirname(dest), exist_ok=True)
48          shutil.copy(src, dest)
49          saved += 1
50      print(f"Baseline updated: {saved} result(s) saved to {BASELINE_DIR}/.")
51
52
53  def check(threshold_pct: float = 20.0) -> bool:
54      """Return True if all benchmarks are within threshold, False otherwise."""
55      if not os.path.isdir(BASELINE_DIR):
56          print(f"No baseline found in {BASELINE_DIR}/.")
57          print("Run 'make bench-baseline' once to establish one, then re-run.")
58          print("Skipping regression check for this run.")
59          return True # don't fail on first-ever run
60
61      threshold = threshold_pct / 100.0
62      failures: list[tuple[str, float]] = []
63      compared = 0
64
65      for cur_path in glob.glob(f"{CRITERION_DIR}/**/current/estimates.json", recursive=True):
66          rel = os.path.relpath(cur_path, CRITERION_DIR)
67          name = rel.split(os.sep)[0]
68          main_path = os.path.join(BASELINE_DIR, rel.replace("/current/", "/main/"))
69          if not os.path.exists(main_path):
70              print(f" (no baseline for {name}, skipping)")
71              continue
72          cur_mean = _load_mean(cur_path)
73          main_mean = _load_mean(main_path)
74          change = (cur_mean - main_mean) / main_mean
75          arrow = "▲" if change > 0 else "▼"
76          print(f" {name}: {arrow}{abs(change) * 100:.1f}%")
77          if change > threshold:

```

```
78     failures.append((name, change))
79     compared += 1
80
81     if compared == 0:
82         print(" (no benchmarks to compare)")
83         return True
84
85     if failures:
86         print(f"\nRegressions exceeding {threshold_pct:.0f}% threshold:")
87         for name, change in failures:
88             print(f" FAIL {name}: +{change * 100:.1f}% slower than baseline")
89         return False
90
91     print("All benchmarks within threshold.")
92     return True
93
94
95 if __name__ == "__main__":
96     if "--save" in sys.argv:
97         save()
98     else:
99         threshold = float(os.getenv("REGRESSION_THRESHOLD", "20"))
100         if not check(threshold):
101             sys.exit(1)
```

## src/cli.rs

```

1 use std::path::PathBuf;
2
3 use clap::Parser;
4
5 use crate::types::{ActivityFilter, PaperSize};
6
7 /// Parsed command-line arguments for the `gitprint` binary.
8 #[derive(Parser, Debug)]
9 #[command(
10     name = "gitprint",
11     about = "Convert git repositories into beautifully formatted PDFs",
12     long_about = "Convert git repositories into beautifully formatted PDFs.\n\
13                 \n\
14                 MODES\n\
15                 \n \
16                 gitprint <PATH> [OPTIONS]\n \
17                 Local path, file, or remote URL (https://, git@, ssh://) → PDF\n\
18                 \n \
19                 gitprint --user <USERNAME> [OPTIONS]\n \
20                 GitHub user activity report → PDF\n\
21                 \n \
22                 gitprint <PATH|--user USERNAME> --preview\n \
23                 Preview output in the terminal – no PDF generated",
24     version,
25     arg_required_else_help = true,
26     after_help = after_help_text(),
27 )]
28 pub struct Args {
29     /// Local path, file, or remote URL (https://, git@, ssh://)
30     pub path: Option<String>,
31
32     /// Preview output in the terminal instead of generating a PDF
33     #[arg(long)]
34     pub preview: bool,
35
36     /// Output PDF file path
37     #[arg(short, long)]
38     pub output: Option<PathBuf>,
39
40     /// — Repository Mode —————
41     /// Glob patterns for files to include (repeatable)
42     #[arg(long, action = clap::ArgAction::Append, help_heading = "Repository Mode (Default)")]
43     pub include: Vec<String>,
44
45     /// Glob patterns for files to exclude (repeatable)
46     #[arg(long, action = clap::ArgAction::Append, help_heading = "Repository Mode (Default)")]
47     pub exclude: Vec<String>,
48
49     /// Syntax highlighting theme
50     #[arg(
51         long,
52         default_value = "InspiredGitHub",
53         help_heading = "Repository Mode (Default)"
54     )]
55     pub theme: String,
56
57     /// Code font size in points
58     #[arg(
59         long,
60         default_value_t = 8.0,
61         help_heading = "Repository Mode (Default)"
62     )]
63     pub font_size: f64,
64
65     /// Disable line numbers
66     #[arg(long, help_heading = "Repository Mode (Default)")]
67     pub no_line_numbers: bool,
68
69     /// Disable table of contents
70     #[arg(long, help_heading = "Repository Mode (Default)")]
71     pub no_toc: bool,
72
73     /// Disable directory tree visualization
74     #[arg(long, help_heading = "Repository Mode (Default)")]
75     pub no_file_tree: bool,
76
77     /// Use a specific branch

```

```
78  #[arg(long, help_heading = "Repository Mode (Default)")]
79  pub branch: Option<String>,
80
81  /// Use a specific commit
82  #[arg(long, help_heading = "Repository Mode (Default)")]
83  pub commit: Option<String>,
84
85  /// Paper size
86  #[arg(long, value_enum, default_value_t = PaperSize::A4, help_heading = "Repository Mode (Default)")]
87  pub paper_size: PaperSize,
88
89  /// Use landscape orientation
90  #[arg(long, help_heading = "Repository Mode (Default)")]
91  pub landscape: bool,
92
93  /// List available syntax themes and exit
94  #[arg(long, help_heading = "Repository Mode (Default)")]
95  pub list_themes: bool,
96
97  /// List version tags of the repository and exit
98  #[arg(long, help_heading = "Repository Mode (Default)")]
99  pub list_tags: bool,
100
101  /// Open the repository in Neovim instead of generating a PDF
102  #[arg(long, help_heading = "Repository Mode (Default)")]
103  pub nvim: bool,
104
105  // — User Report Mode —————
106  /// GitHub username – generate a user activity report instead of printing a repo
107  #[arg(short = 'u', long = "user", help_heading = "User Report Mode")]
108  pub user: Option<String>,
109
110  /// Number of most-recently-pushed repos to include [default: 5]
111  #[arg(long, default_value_t = 5, help_heading = "User Report Mode")]
112  pub last_repos: usize,
113
114  /// Number of recent commits with diffs to render [default: 5]
115  #[arg(long, default_value_t = 5, help_heading = "User Report Mode")]
116  pub last_commits: usize,
117
118  /// Skip commit diff rendering (faster)
119  #[arg(long, help_heading = "User Report Mode")]
120  pub no_diffs: bool,
121
122  /// Show events from this date forward [default: no lower bound; GitHub keeps ≤ 90 days]
123  ///
124  /// Accepted formats:
125  ///   ISO date      2024-01-15 or 2024-01-15T00:00:00Z
126  ///   Keywords     today · yesterday
127  ///   Named        last week · last month · last year
128  ///   Relative     30 days ago · 2 weeks ago · 1 month ago · 1 year ago
129  #[arg(long, value_name = "DATE", help_heading = "User Report Mode")]
130  pub since: Option<String>,
131
132  /// Show events up to and including this date [default: no upper bound]
133  ///
134  /// Same formats as --since.
135  #[arg(long, value_name = "DATE", help_heading = "User Report Mode")]
136  pub until: Option<String>,
137
138  /// Event types to include in the activity feed [default: all]
139  ///
140  /// all           – every public event (pushes, PRs, issues, stars, forks, ...)
141  /// commits       – push events only
142  #[arg(long, value_enum, default_value_t = ActivityFilter::All, help_heading = "User Report Mode")]
143  pub activity: ActivityFilter,
144
145  /// Maximum events shown in the activity feed [default: 30]
146  ///
147  /// Fetches up to 100 events from GitHub and applies --since/--until/--activity
148  /// filters before counting toward this limit.
149  #[arg(long, default_value_t = 30, help_heading = "User Report Mode")]
150  pub events: usize,
151 }
152
153 fn after_help_text() -> &'static str {
154     static TEXT: std::sync::OnceLock<String> = std::sync::OnceLock::new();
155     TEXT.get_or_init(|| {
```

```

156     let version = match option_env!("GITPRINT_GIT_BRANCH") {
157         Some(branch) => branch.to_string(),
158         None => format!("{}", env!("CARGO_PKG_VERSION")),
159     };
160     let size_str = std::env::current_exe()
161         .ok()
162         .and_then(|p| std::fs::metadata(p).ok())
163         .map(|m| {
164             let bytes = m.len();
165             let (size, unit) = if bytes >= 1_048_576 {
166                 (bytes as f64 / 1_048_576.0, "MB")
167             } else {
168                 (bytes as f64 / 1_024.0, "KB")
169             };
170             format!("{size:.1} {unit}")
171         })
172         .unwrap_or_else(|| "unknown".to_string());
173     format!("Version: {version} | Binary size: {size_str}\nSponsor: https://github.com/sponsors/izelnakri")
174 }
175 }
176
177 #[cfg(test)]
178 mod tests {
179     use super::*;
180     use clap::Parser;
181
182     #[test]
183     fn requires_path_or_user() {
184         assert!(Args::try_parse_from(["gitprint"]).is_err());
185     }
186
187     #[test]
188     fn accepts_path() {
189         let args = Args::parse_from(["gitprint", "."]);
190         assert_eq!(args.path, Some(".").to_string());
191     }
192
193     #[test]
194     fn custom_path() {
195         let args = Args::parse_from(["gitprint", "/tmp/repo"]);
196         assert_eq!(args.path, Some("/tmp/repo".to_string()));
197     }
198
199     #[test]
200     fn accepts_https_url() {
201         let args = Args::parse_from(["gitprint", "https://github.com/user/repo"]);
202         assert_eq!(args.path, Some("https://github.com/user/repo".to_string()));
203     }
204
205     #[test]
206     fn accepts_ssh_url() {
207         let args = Args::parse_from(["gitprint", "git@github.com:user/repo.git"]);
208         assert_eq!(args.path, Some("git@github.com:user/repo.git".to_string()));
209     }
210
211     #[test]
212     fn user_flag_short() {
213         let args = Args::parse_from(["gitprint", "-u", "izelnakri"]);
214         assert_eq!(args.user, Some("izelnakri".to_string()));
215         assert_eq!(args.path, None);
216     }
217
218     #[test]
219     fn user_flag_long() {
220         let args = Args::parse_from(["gitprint", "--user", "torvalds"]);
221         assert_eq!(args.user, Some("torvalds".to_string()));
222     }
223
224     #[test]
225     fn user_flag_with_output() {
226         let args = Args::parse_from(["gitprint", "-u", "alice", "-o", "alice.pdf"]);
227         assert_eq!(args.user, Some("alice".to_string()));
228         assert_eq!(args.output, Some(PathBuf::from("alice.pdf")));
229     }
230
231     #[test]
232     fn user_report_flags_defaults() {
233         let args = Args::parse_from(["gitprint", "-u", "alice"]);

```

```
234     assert_eq!(args.last_repos, 5);
235     assert_eq!(args.last_commits, 5);
236     assert!(!args.no_diffs);
237     assert_eq!(args.events, 30);
238     assert!(matches!(args.activity, ActivityFilter::All));
239     assert!(args.since.is_none());
240     assert!(args.until.is_none());
241 }
242
243 #[test]
244 fn since_until_flags() {
245     let args = Args::parse_from(["gitprint", "-u", "alice", "--since", "2024-01-01"]);
246     assert_eq!(args.since.as_deref(), Some("2024-01-01"));
247     let args = Args::parse_from([
248         "gitprint",
249         "-u",
250         "alice",
251         "--since",
252         "30 days ago",
253         "--until",
254         "yesterday",
255     ]);
256     assert_eq!(args.since.as_deref(), Some("30 days ago"));
257     assert_eq!(args.until.as_deref(), Some("yesterday"));
258 }
259
260 #[test]
261 fn activity_flag() {
262     let args = Args::parse_from(["gitprint", "-u", "alice", "--activity", "commits"]);
263     assert!(matches!(args.activity, ActivityFilter::Commits));
264     let args = Args::parse_from(["gitprint", "-u", "alice", "--activity", "all"]);
265     assert!(matches!(args.activity, ActivityFilter::All));
266 }
267
268 #[test]
269 fn events_flag() {
270     let args = Args::parse_from(["gitprint", "-u", "alice", "--events", "50"]);
271     assert_eq!(args.events, 50);
272 }
273
274 #[test]
275 fn user_report_flags_custom() {
276     let args = Args::parse_from([
277         "gitprint",
278         "-u",
279         "alice",
280         "--last-commits",
281         "3",
282         "--no-diffs",
283     ]);
284     assert_eq!(args.last_commits, 3);
285     assert!(args.no_diffs);
286 }
287
288 #[test]
289 fn output_short_flag() {
290     let args = Args::parse_from(["gitprint", ".", "-o", "out.pdf"]);
291     assert_eq!(args.output, Some(PathBuf::from("out.pdf")));
292 }
293
294 #[test]
295 fn output_long_flag() {
296     let args = Args::parse_from(["gitprint", ".", "--output", "out.pdf"]);
297     assert_eq!(args.output, Some(PathBuf::from("out.pdf")));
298 }
299
300 #[test]
301 fn all_flags() {
302     let args = Args::parse_from([
303         "gitprint",
304         "https://github.com/user/repo",
305         "-o",
306         "out.pdf",
307         "--theme",
308         "Solarized (dark)",
309         "--font-size",
310         "10",
311         "--no-line-numbers",
```

```
312     "--no-toc",
313     "--no-file-tree",
314     "--branch",
315     "dev",
316     "--paper-size",
317     "letter",
318     "--landscape",
319     "--list-themes",
320 ];
321 assert_eq!(args.path, Some("https://github.com/user/repo".to_string()));
322 assert_eq!(args.output, Some(PathBuf::from("out.pdf")));
323 assert_eq!(args.theme, "Solarized (dark)");
324 assert_eq!(args.font_size, 10.0);
325 assert!(args.no_line_numbers);
326 assert!(args.no_toc);
327 assert!(args.no_file_tree);
328 assert_eq!(args.branch, Some("dev".to_string()));
329 assert!(matches!(args.paper_size, PaperSize::Letter));
330 assert!(args.landscape);
331 assert!(args.list_themes);
332 }
333
334 #[test]
335 fn list_tags_flag() {
336     let args = Args::parse_from(["gitprint", ".", "--list-tags"]);
337     assert!(args.list_tags);
338     let args = Args::parse_from(["gitprint", "."]);
339     assert!(!args.list_tags);
340 }
341
342 #[test]
343 fn commit_flag() {
344     let args = Args::parse_from(["gitprint", ".", "--commit", "abc1234"]);
345     assert_eq!(args.commit, Some("abc1234".to_string()));
346 }
347
348 #[test]
349 fn paper_size_legal() {
350     let args = Args::parse_from(["gitprint", ".", "--paper-size", "legal"]);
351     assert!(matches!(args.paper_size, PaperSize::Legal));
352 }
353
354 #[test]
355 fn multiple_include_exclude() {
356     let args = Args::parse_from([
357         "gitprint",
358         ".",
359         "--include",
360         "*.rs",
361         "--include",
362         "*.toml",
363         "--exclude",
364         "*.lock",
365         "--exclude",
366         "*.md",
367     ]);
368     assert_eq!(args.include, vec!["*.rs", "*.toml"]);
369     assert_eq!(args.exclude, vec!["*.lock", "*.md"]);
370 }
371
372 #[test]
373 fn font_size_custom() {
374     let args = Args::parse_from(["gitprint", ".", "--font-size", "12.5"]);
375     assert_eq!(args.font_size, 12.5);
376 }
377
378 #[test]
379 fn preview_flag() {
380     let args = Args::parse_from(["gitprint", ".", "--preview"]);
381     assert!(args.preview);
382     let args = Args::parse_from(["gitprint", "."]);
383     assert!(!args.preview);
384 }
385
386 #[test]
387 fn preview_with_user() {
388     let args = Args::parse_from(["gitprint", "-u", "alice", "--preview"]);
389     assert!(args.preview);

```

```
390     assert_eq!(args.user, Some("alice".to_string()));
391 }
392 }
```

## src/defaults.rs

```

1  /// Default glob patterns excluded from PDF output (lock files, build artifacts, binaries, etc.).
2  pub const DEFAULT_EXCLUDES: &[&str] = &[
3      // Lock files
4      "package-lock.json",
5      "yarn.lock",
6      "pnpm-lock.yaml",
7      "Cargo.lock",
8      "Gemfile.lock",
9      "composer.lock",
10     "poetry.lock",
11     "Pipfile.lock",
12     "flake.lock",
13     // Build output
14     "node_modules/**",
15     "target/**",
16     "dist/**",
17     "build/**",
18     ".next/**",
19     "__pycache__/**",
20     "*.pyc",
21     // VCS / IDE
22     ".git/**",
23     ".svn/**",
24     ".idea/**",
25     ".vscode/**",
26     "*.swp",
27     "*.swo",
28     ".DS_Store",
29     // Images
30     "*.png",
31     "*.jpg",
32     "*.jpeg",
33     "*.gif",
34     "*.ico",
35     "*.svg",
36     "*.webp",
37     "*.bmp",
38     // Fonts
39     "*.woff",
40     "*.woff2",
41     "*.ttf",
42     "*.otf",
43     "*.eot",
44     // Archives / binaries
45     "*.zip",
46     "*.tar",
47     "*.gz",
48     "*.bz2",
49     "*.xz",
50     "*.7z",
51     "*.rar",
52     "*.exe",
53     "*.dll",
54     "*.so",
55     "*.dylib",
56     "*.o",
57     "*.a",
58     "*.class",
59     "*.jar",
60     "*.war",
61     "*.wasm",
62     // Generated / minified
63     "*.min.js",
64     "*.min.css",
65     "*.map",
66     "*.bundle.js",
67     // Data
68     "*.sqlite",
69     "*.db",
70     "*.pdf",
71 ];
72
73 #[cfg(test)]
74 mod tests {
75     use super::*;
76     use globset::Glob;
77

```

```
78 #[test]
79 fn default_excludes_has_entries() {
80     assert!(DEFAULT_EXCLUDES.len() > 10);
81 }
82
83 #[test]
84 fn all_patterns_are_valid_globs() {
85     DEFAULT_EXCLUDES.iter().for_each(|pattern| {
86         Glob::new(pattern).unwrap_or_else(|e| panic!("invalid glob '{pattern}': {e}"));
87     });
88 }
89
90 #[test]
91 fn known_lock_files_present() {
92     assert!(DEFAULT_EXCLUDES.contains(&"Cargo.lock"));
93     assert!(DEFAULT_EXCLUDES.contains(&"package-lock.json"));
94     assert!(DEFAULT_EXCLUDES.contains(&"yarn.lock"));
95     assert!(DEFAULT_EXCLUDES.contains(&"poetry.lock"));
96     assert!(DEFAULT_EXCLUDES.contains(&"flake.lock"));
97 }
98
99 #[test]
100 fn known_build_dirs_present() {
101     assert!(DEFAULT_EXCLUDES.contains(&"node_modules/**"));
102     assert!(DEFAULT_EXCLUDES.contains(&"target/**"));
103     assert!(DEFAULT_EXCLUDES.contains(&"dist/**"));
104     assert!(DEFAULT_EXCLUDES.contains(&"__pycache__/**"));
105 }
106
107 #[test]
108 fn known_image_extensions_present() {
109     assert!(DEFAULT_EXCLUDES.contains(&"*.png"));
110     assert!(DEFAULT_EXCLUDES.contains(&"*.jpg"));
111     assert!(DEFAULT_EXCLUDES.contains(&"*.svg"));
112 }
113
114 #[test]
115 fn known_binary_extensions_present() {
116     assert!(DEFAULT_EXCLUDES.contains(&"*.exe"));
117     assert!(DEFAULT_EXCLUDES.contains(&"*.wasm"));
118     assert!(DEFAULT_EXCLUDES.contains(&"*.zip"));
119 }
120
121 #[test]
122 fn known_generated_extensions_present() {
123     assert!(DEFAULT_EXCLUDES.contains(&"*.min.js"));
124     assert!(DEFAULT_EXCLUDES.contains(&"*.min.css"));
125     assert!(DEFAULT_EXCLUDES.contains(&"*.map"));
126 }
127
128 #[test]
129 fn vcs_and_ide_dirs_present() {
130     assert!(DEFAULT_EXCLUDES.contains(&".git/**"));
131     assert!(DEFAULT_EXCLUDES.contains(&".idea/**"));
132     assert!(DEFAULT_EXCLUDES.contains(&".vscode/**"));
133     assert!(DEFAULT_EXCLUDES.contains(&".DS_Store"));
134 }
135 }
```

## src/filter.rs

```

1 use std::path::{Path, PathBuf};
2
3 use globset::{Glob, GlobSet, GlobSetBuilder};
4
5 use crate::defaults::DEFAULT_EXCLUDES;
6
7 /// Filters file paths based on glob include/exclude patterns.
8 ///
9 /// Exclude patterns always take precedence over include patterns.
10 /// Default excludes (lock files, binaries, build artifacts) are always applied.
11 pub struct FileFilter {
12     include_set: Option<GlobSet>,
13     exclude_set: GlobSet,
14 }
15
16 impl FileFilter {
17     /// Creates a new `FileFilter` from glob include and exclude patterns.
18     ///
19     /// An empty `include_patterns` slice allows all files (subject to excludes).
20     /// Default excludes (lock files, build artifacts, binaries, etc.) are always applied.
21     ///
22     /// # Errors
23     ///
24     /// Returns an error if any glob pattern is invalid.
25     ///
26     /// # Examples
27     ///
28     /// ```
29     /// use gitprint::filter::FileFilter;
30     /// use std::path::Path;
31     ///
32     /// // Include only Rust files, exclude test helpers
33     /// let filter = FileFilter::new(
34     ///     &["*.rs".to_string()],
35     ///     &["test_*.rs".to_string()],
36     /// ).unwrap();
37     ///
38     /// assert!(filter.should_include(Path::new("main.rs")));
39     /// assert!(!filter.should_include(Path::new("test_helper.rs")));
40     /// assert!(!filter.should_include(Path::new("README.md")));
41     /// ```
42     pub fn new(include_patterns: &[String], exclude_patterns: &[String]) -> anyhow::Result<Self> {
43         let include_set = if include_patterns.is_empty() {
44             None
45         } else {
46             let set = include_patterns
47                 .iter()
48                 .try_fold(GlobSetBuilder::new(), |mut b, p| {
49                     b.add(
50                         Glob::new(p)
51                             .map_err(|e| anyhow::anyhow!("invalid glob pattern '{p}': {e}"))?,
52                     );
53                     Ok:::<_, anyhow::Error>(b)
54                 })?
55                 .build()
56                 .map_err(|e| anyhow::anyhow!("failed to build glob set: {e}"))?;
57             Some(set)
58         };
59
60         let exclude_set = DEFAULT_EXCLUDES
61             .iter()
62             .map(|p| Glob::new(p).unwrap())
63             .chain(
64                 exclude_patterns
65                     .iter()
66                     .map(|p| {
67                         Glob::new(p).map_err(|e| anyhow::anyhow!("invalid glob pattern '{p}': {e}"))
68                     })
69             )
70             .collect:::<anyhow::Result<Vec<_>>>()?
71             .into_iter(),
72         )
73         .fold(GlobSetBuilder::new(), |mut b, g| {
74             b.add(g);
75             b
76         })
77         .build()
78         .map_err(|e| anyhow::anyhow!("failed to build glob set: {e}"))?;

```

```

78
79     Ok(Self {
80         include_set,
81         exclude_set,
82     })
83 }
84
85 /// Returns `true` if `path` should be included given the configured patterns.
86 ///
87 /// Exclude patterns always win over include patterns.
88 ///
89 /// # Examples
90 ///
91 /// ```
92 /// use gitprint::filter::FileFilter;
93 /// use std::path::Path;
94 ///
95 /// let filter = FileFilter::new(&["*.rs".to_string()], &[]).unwrap();
96 /// assert!(filter.should_include(Path::new("src/lib.rs")));
97 /// assert!(!filter.should_include(Path::new("Cargo.toml")));
98 /// assert!(!filter.should_include(Path::new("Cargo.lock"))); // default exclude
99 /// ```
100 pub fn should_include(&self, path: &Path) -> bool {
101     if self.exclude_set.is_match(path) {
102         return false;
103     }
104     self.include_set
105         .as_ref()
106         .is_none_or(|set| set.is_match(path))
107 }
108
109 /// Filters a list of paths, retaining only those that pass `should_include`.
110 ///
111 /// # Examples
112 ///
113 /// ```
114 /// use gitprint::filter::FileFilter;
115 /// use std::path::PathBuf;
116 ///
117 /// let filter = FileFilter::new(&["*.rs".to_string()], &[]).unwrap();
118 /// let paths = vec![
119 ///     PathBuf::from("main.rs"),
120 ///     PathBuf::from("README.md"),
121 ///     PathBuf::from("lib.rs"),
122 /// ];
123 /// let kept: Vec<_> = filter.filter_paths(paths).collect();
124 /// assert_eq!(kept, vec![PathBuf::from("main.rs"), PathBuf::from("lib.rs")]);
125 /// ```
126 pub fn filter_paths(&self, paths: Vec<PathBuf>) -> impl Iterator<Item = PathBuf> + '_ {
127     paths.into_iter().filter(|p| self.should_include(p))
128 }
129 }
130
131 /// Returns `true` if the content appears to be a binary file.
132 ///
133 /// Detection is based on the presence of non-text byte sequences (e.g. null bytes).
134 ///
135 /// # Examples
136 ///
137 /// ```
138 /// use gitprint::filter::is_binary;
139 ///
140 /// assert!(is_binary(b"hello\x00world")); // null byte → binary
141 /// assert!(!is_binary(b"fn main() {}")); // valid UTF-8 → not binary
142 /// assert!(!is_binary(b""));
143 /// ```
144 pub fn is_binary(content: &[u8]) -> bool {
145     content.iter().take(8000).any(|&b| b == 0)
146 }
147
148 /// Returns `true` if the content appears to be minified.
149 ///
150 /// A file is considered minified when any of its first 5 lines exceeds 500 characters,
151 /// which is characteristic of bundled or minified JavaScript/CSS.
152 ///
153 /// # Examples
154 ///
155 /// ```

```

```
156 /// use gitprint::filter::is_minified;
157 ///
158 /// assert!(is_minified(&"x".repeat(501))); // single very long line
159 /// assert!(!is_minified("fn main() {\n  println!(\"hello\");\n}\n"));
160 /// assert!(!is_minified(""));
161 /// ```
162 pub fn is_minified(content: &str) -> bool {
163     content.lines().take(5).any(|line| line.len() > 500)
164 }
165
166 #[cfg(test)]
167 mod tests {
168     use super::*;
169
170     #[test]
171     fn default_excludes_applied() {
172         let filter = FileFilter::new(&[], &[]).unwrap();
173         assert!(!filter.should_include(Path::new("Cargo.lock")));
174         assert!(!filter.should_include(Path::new("node_modules/foo.js")));
175         assert!(!filter.should_include(Path::new("image.png")));
176         assert!(!filter.should_include(Path::new("target/debug/binary")));
177         assert!(!filter.should_include(Path::new(".git/HEAD")));
178         assert!(!filter.should_include(Path::new("bundle.min.js")));
179     }
180
181     #[test]
182     fn custom_exclude() {
183         let filter = FileFilter::new(&[], &["*.md".to_string()]).unwrap();
184         assert!(!filter.should_include(Path::new("README.md")));
185         assert!(!filter.should_include(Path::new("docs/GUIDE.md")));
186         assert!(filter.should_include(Path::new("main.rs")));
187     }
188
189     #[test]
190     fn include_only() {
191         let filter = FileFilter::new(&["*.rs".to_string()], &[]).unwrap();
192         assert!(filter.should_include(Path::new("main.rs")));
193         assert!(filter.should_include(Path::new("src/lib.rs")));
194         assert!(!filter.should_include(Path::new("README.md")));
195         assert!(!filter.should_include(Path::new("Cargo.toml")));
196     }
197
198     #[test]
199     fn include_and_exclude_interaction() {
200         let filter = FileFilter::new(&["*.rs".to_string()], &["test_*.rs".to_string()]).unwrap();
201         assert!(filter.should_include(Path::new("main.rs")));
202         assert!(!filter.should_include(Path::new("test_helper.rs")));
203     }
204
205     #[test]
206     fn empty_filter_includes_normal_files() {
207         let filter = FileFilter::new(&[], &[]).unwrap();
208         assert!(filter.should_include(Path::new("src/main.rs")));
209         assert!(filter.should_include(Path::new("Cargo.toml")));
210         assert!(filter.should_include(Path::new("README.md")));
211     }
212
213     #[test]
214     fn multiple_include_patterns() {
215         let filter = FileFilter::new(&["*.rs".to_string(), "*.toml".to_string()], &[]).unwrap();
216         assert!(filter.should_include(Path::new("main.rs")));
217         assert!(filter.should_include(Path::new("Cargo.toml")));
218         assert!(!filter.should_include(Path::new("README.md")));
219     }
220
221     #[test]
222     fn multiple_exclude_patterns() {
223         let filter = FileFilter::new(&[], &["*.md".to_string(), "*.txt".to_string()]).unwrap();
224         assert!(!filter.should_include(Path::new("README.md")));
225         assert!(!filter.should_include(Path::new("notes.txt")));
226         assert!(filter.should_include(Path::new("main.rs")));
227     }
228
229     #[test]
230     fn exclude_takes_precedence_over_include() {
231         let filter = FileFilter::new(&["*.rs".to_string()], &["main.rs".to_string()]).unwrap();
232         assert!(!filter.should_include(Path::new("main.rs")));
233         assert!(filter.should_include(Path::new("lib.rs")));
234     }
235 }
```

```
234 }
235
236 #[test]
237 fn filter_paths_works() {
238     let filter = FileFilter::new(&["*.rs".to_string()], &[]).unwrap();
239     let paths = vec![
240         PathBuf::from("main.rs"),
241         PathBuf::from("README.md"),
242         PathBuf::from("lib.rs"),
243     ];
244     let filtered: Vec<_> = filter.filter_paths(paths).collect();
245     assert_eq!(
246         filtered,
247         vec![PathBuf::from("main.rs"), PathBuf::from("lib.rs")]
248     );
249 }
250
251 #[test]
252 fn filter_paths_empty_input() {
253     let filter = FileFilter::new(&[], &[]).unwrap();
254     let filtered: Vec<_> = filter.filter_paths(vec![]).collect();
255     assert!(filtered.is_empty());
256 }
257
258 #[test]
259 fn is_binary_with_null_bytes() {
260     let content = b"hello\x00world";
261     assert!(is_binary(content));
262 }
263
264 #[test]
265 fn is_binary_with_text() {
266     let content = b"fn main() { println!(\"hello\"); }";
267     assert!(!is_binary(content));
268 }
269
270 #[test]
271 fn is_binary_with_empty() {
272     assert!(!is_binary(b""));
273 }
274
275 #[test]
276 fn is_binary_with_utf8() {
277     assert!(!is_binary("0 0 0 0 0 0 ".as_bytes()));
278 }
279
280 #[test]
281 fn is_minified_with_long_line() {
282     let long_line = "a".repeat(501);
283     assert!(is_minified(&long_line));
284 }
285
286 #[test]
287 fn is_minified_with_normal_content() {
288     assert!(!is_minified("fn main() {\n    println!(\"hi\");\n}\n"));
289 }
290
291 #[test]
292 fn is_minified_long_line_after_fifth() {
293     let mut content = "short\n".repeat(5);
294     content.push_str(&"a".repeat(501));
295     assert!(!is_minified(&content));
296 }
297
298 #[test]
299 fn is_minified_exactly_500_chars() {
300     let line = "a".repeat(500);
301     assert!(!is_minified(&line));
302 }
303
304 #[test]
305 fn is_minified_empty() {
306     assert!(!is_minified(""));
307 }
308
309 #[test]
310 fn is_minified_long_line_on_line_3() {
311     let content = format!("short\nshort\n{}\nshort\nshort\n", "a".repeat(501));
```

```
312     assert!(is_minified(&content));
313 }
314
315 #[test]
316 fn invalid_include_glob_returns_error() {
317     let result = FileFilter::new(&["invalid".to_string()], &[]);
318     assert!(result.is_err());
319 }
320
321 #[test]
322 fn invalid_exclude_glob_returns_error() {
323     let result = FileFilter::new(&[], &["invalid".to_string()]);
324     assert!(result.is_err());
325 }
326 }
```

## src/git.rs

```

1 use std::collections::HashMap;
2 use std::future::Future;
3 use std::path::{Path, PathBuf};
4 use std::pin::Pin;
5 use std::sync::Arc;
6 use std::time::UNIX_EPOCH;
7
8 use anyhow::bail;
9 use tokio::process::Command;
10
11 use crate::types::{Config, RepoMetadata};
12
13 /// Returns `true` if `s` looks like a remote git URL.
14 ///
15 /// Recognised schemes: `https://`, `http://`, `git://`, `ssh://`,
16 /// and SCP-style `git@host:path` used by GitHub/GitLab.
17 pub fn is_remote_url(s: &str) -> bool {
18     s.starts_with("https://")
19     || s.starts_with("http://")
20     || s.starts_with("git://")
21     || s.starts_with("ssh://")
22     || (s.contains('@') && s.contains(':') && !s.starts_with('/'))
23 }
24
25 /// Extracts the repository name from a remote URL.
26 ///
27 /// `https://github.com/user/repo.git` → `repo`
28 /// `git@github.com:user/repo` → `repo`
29 pub fn repo_name_from_url(url: &str) -> String {
30     url.split(['/', ':'])
31         .next_back()
32         .unwrap_or("repo")
33         .trim_end_matches(".git")
34         .to_string()
35 }
36
37 /// A temporary directory that deletes itself on drop.
38 pub struct TempCloneDir(PathBuf);
39
40 impl TempCloneDir {
41     /// Creates (or reuses) a deterministically-named temp dir based on the URL,
42     /// branch, and commit so repeated invocations for the same target don't
43     /// accumulate stale copies in `/tmp`.
44     pub async fn for_url(
45         url: &str,
46         branch: Option<&str>,
47         commit: Option<&str>,
48     ) -> anyhow::Result<Self> {
49         use std::collections::hash_map::DefaultHasher;
50         use std::hash::{Hash, Hasher};
51         let mut h = DefaultHasher::new();
52         url.hash(&mut h);
53         branch.hash(&mut h);
54         commit.hash(&mut h);
55         let dir = std::env::temp_dir().join(format!("gitprint-{:016x}", h.finish()));
56         tokio::fs::create_dir_all(&dir).await?;
57         Ok(Self(dir))
58     }
59
60     /// Returns the path to the temporary clone directory.
61     pub fn path(&self) -> &Path {
62         &self.0
63     }
64 }
65
66 impl Drop for TempCloneDir {
67     fn drop(&mut self) {
68         // Drop is synchronous by design – tokio async cannot be used here.
69         let _ = std::fs::remove_dir_all(&self.0);
70     }
71 }
72
73 /// Clones a remote git repository into `dest`.
74 ///
75 /// Uses `--depth=1` (shallow) for speed unless `commit` is specified, in which
76 /// case a full clone is required to access arbitrary history.
77 pub async fn clone_repo(

```

```

78     url: &str,
79     dest: &Path,
80     branch: Option<&str>,
81     commit: Option<&str>,
82 ) -> anyhow::Result<()> {
83     let mut cmd = Command::new("git");
84     cmd.arg("clone");
85
86     if commit.is_none() {
87         cmd.arg("--depth=1");
88         if let Some(b) = branch {
89             cmd.args(["--branch", b, "--single-branch"]);
90         }
91     } else if let Some(b) = branch {
92         cmd.args(["--branch", b]);
93     }
94
95     let status = cmd
96         .arg(url)
97         .arg(dest)
98         .stderr(std::process::Stdio::inherit())
99         .status()
100        .await
101        .map_err(|e| anyhow::anyhow!("failed to run git: {e}"))?;
102
103     if !status.success() {
104         bail!("git clone failed for {url}");
105     }
106     Ok(())
107 }
108
109 async fn run_git(repo_path: &Path, args: &[&str]) -> anyhow::Result<String> {
110     let output = Command::new("git")
111         .args(["-C", &repo_path.to_string_lossy()])
112         .args(args)
113         .output()
114         .await
115         .map_err(|e| anyhow::anyhow!("failed to run git: {e}"))?;
116
117     if !output.status.success() {
118         let stderr = String::from_utf8_lossy(&output.stderr);
119         bail!("{}, stderr.trim()", stderr);
120     }
121
122     Ok(String::from_utf8(output.stdout)
123         .unwrap_or_else(|e| String::from_utf8_lossy(e.as_bytes()).into_owned()))
124 }
125
126 /// Describes what the user-supplied path resolves to.
127 #[derive(Debug)]
128 pub struct RepoInfo {
129     /// Git repo root (git mode) or canonical directory path (plain-dir mode).
130     pub root: PathBuf,
131     /// Whether `root` is inside a git repository.
132     pub is_git: bool,
133     /// Subdirectory scope within the git repo (relative to `root`).
134     /// Only set when the user supplied a strict subdirectory of the repo root.
135     pub scope: Option<PathBuf>,
136     /// When the user supplied a single file, its path relative to `root`.
137     pub single_file: Option<PathBuf>,
138 }
139
140 /// Resolves a user-supplied path into a [RepoInfo].
141 ///
142 /// Handles four cases:
143 ///
144 /// - File inside a git repo → `single_file` is set, `root` is the repo root.
145 /// - Subdirectory inside a git repo → `scope` is set relative to `root`.
146 /// - Git repo root → `root` is the repo root, no scope.
147 /// - Plain directory or file outside git → `is_git` is `false`.
148 ///
149 /// # Errors
150 ///
151 /// Returns an error if the path does not exist.
152 ///
153 /// # Examples
154 ///
155 /// ``ignore

```

```

156 /// use gitprint::git::verify_repo;
157 /// use std::path::Path;
158 ///
159 /// let info = verify_repo(Path::new(".")).await.unwrap();
160 /// println!("repo root: {}", info.root.display());
161 /// println!("is git: {}", info.is_git);
162 /// ```
163 pub async fn verify_repo(path: &Path) -> anyhow::Result<RepoInfo> {
164     // Use async canonicalize to avoid blocking tokio worker threads.
165     let canonical = tokio::fs::canonicalize(path)
166         .await
167         .map_err(|_| anyhow::anyhow!("{}: path not found", path.display()))?;
168
169     // Fetch metadata once (async stat) and reuse is_file/is_dir throughout -
170     // avoids multiple blocking stat() calls on the same already-resolved path.
171     let meta = tokio::fs::metadata(&canonical)
172         .await
173         .map_err(|_| anyhow::anyhow!("{}: cannot stat path", canonical.display()))?;
174     let is_file = meta.is_file();
175     let is_dir = meta.is_dir();
176
177     // Git must be invoked from a directory; use parent when the path is a file.
178     let git_dir = if is_file {
179         canonical
180             .parent()
181             .ok_or_else(|_| anyhow::anyhow!("file has no parent directory"))?
182             .to_path_buf()
183     } else {
184         canonical.clone()
185     };
186
187     let output = Command::new("git")
188         .args(["-C", &git_dir.to_string_lossy()])
189         .args(["rev-parse", "--show-toplevel"])
190         .output()
191         .await
192         .map_err(|e| anyhow::anyhow!("failed to run git: {e}"))?;
193
194     if output.status.success() {
195         let root = PathBuf::from(String::from_utf8_lossy(&output.stdout).trim().to_string());
196
197         if is_file {
198             let rel = canonical
199                 .strip_prefix(&root)
200                 .map_err(|_| anyhow::anyhow!("file is outside the git repository"))?
201                 .to_path_buf();
202             return Ok(RepoInfo {
203                 root,
204                 is_git: true,
205                 scope: None,
206                 single_file: Some(rel),
207             });
208         }
209
210         let scope = (canonical != root)
211             .then(|_| canonical.strip_prefix(&root).ok().map(|p| p.to_path_buf()))
212             .flatten();
213         return Ok(RepoInfo {
214             root,
215             is_git: true,
216             scope,
217             single_file: None,
218         });
219     }
220
221     // Not inside a git repo.
222     if is_file {
223         let parent = canonical
224             .parent()
225             .ok_or_else(|_| anyhow::anyhow!("file has no parent directory"))?
226             .to_path_buf();
227         return Ok(RepoInfo {
228             root: parent,
229             is_git: false,
230             scope: None,
231             single_file: Some(PathBuf::from(canonical.file_name().unwrap())),
232         });
233     }

```

```

234
235     if is_dir {
236         return Ok(RepoInfo {
237             root: canonical,
238             is_git: false,
239             scope: None,
240             single_file: None,
241         });
242     }
243
244     bail!(
245         "{}: not a git repository, directory, or file",
246         path.display()
247     )
248 }
249
250 /// Fetches repository metadata: branch, last commit hash/date/message, and name.
251 ///
252 /// For non-git directories, returns a `RepoMetadata` with empty git fields.
253 /// Branch detection and commit log are fetched concurrently.
254 ///
255 /// # Errors
256 ///
257 /// Returns an error if the git command fails (git repos only).
258 pub async fn get_metadata(
259     repo_path: &Path,
260     config: &Config,
261     is_git: bool,
262     scope: Option<&Path>,
263 ) -> anyhow::Result<RepoMetadata> {
264     let base = repo_path
265         .file_name()
266         .map(|n| n.to_string_lossy().to_string())
267         .unwrap_or_else(|| "unknown".to_string());
268     let name = match scope {
269         Some(s) => format!("{}", s.display()),
270         None => base,
271     };
272
273     if !is_git {
274         return Ok(RepoMetadata {
275             name,
276             branch: String::new(),
277             commit_hash: String::new(),
278             commit_hash_short: String::new(),
279             commit_date: String::new(),
280             commit_message: String::new(),
281             commit_author: String::new(),
282             commit_author_email: String::new(),
283             file_count: 0,
284             total_lines: 0,
285             fs_owner: None,
286             fs_group: None,
287             generated_at: String::new(),
288             repo_size: String::new(),
289             fs_size: String::new(),
290             detected_remote_url: None,
291             repo_absolute_path: None,
292         });
293     }
294
295     let rev = match (&config.commit, &config.branch) {
296         (Some(c), _) => c.clone(),
297         (_, Some(b)) => b.clone(),
298         _ => "HEAD".to_string(),
299     };
300
301     // Run branch detection, commit log, and remote URL detection in parallel.
302     // Format: hash, date, subject, author name, author email (one per line, %n separated).
303     let log_args = ["log", "-1", "--format=%H%n%ci%n%s%n%an%n%ae", &rev];
304     let (branch, log_output, detected_remote_url) = tokio::join!(
305         async {
306             match &config.branch {
307                 Some(b) => b.clone(),
308                 None => run_git(repo_path, &["rev-parse", "--abbrev-ref", "HEAD"])
309                     .await
310                     .map(|s| s.trim().to_string())
311                     .unwrap_or_else(|_| "detached".to_string()),

```

```

312     }
313   },
314   run_git(repo_path, &log_args),
315   git_remote_url(repo_path),
316 );
317 let log_output = log_output?;
318
319 let mut lines = log_output.trim().lines();
320 let commit_hash = lines.next().unwrap_or("").to_string();
321 let commit_hash_short = commit_hash[..7.min(commit_hash.len())].to_string();
322 let commit_date = lines.next().unwrap_or("").to_string();
323 // Remaining: subject lines, then author name, then author email (last two lines).
324 let remaining: Vec<&str> = lines.collect();
325 let (commit_message, commit_author, commit_author_email) = match remaining.as_slice() {
326     [] => (String::new(), String::new(), String::new()),
327     [.., author, email] => {
328         let subject_lines = &remaining[..remaining.len().saturating_sub(2)];
329         (
330             subject_lines.join("\n"),
331             author.to_string(),
332             email.to_string(),
333         )
334     }
335     [author] => (String::new(), author.to_string(), String::new()),
336 };
337
338 Ok(RepoMetadata {
339     name,
340     branch,
341     commit_hash,
342     commit_hash_short,
343     commit_date,
344     commit_message,
345     commit_author,
346     commit_author_email,
347     file_count: 0,
348     total_lines: 0,
349     fs_owner: None,
350     fs_group: None,
351     generated_at: String::new(),
352     repo_size: String::new(),
353     fs_size: String::new(),
354     detected_remote_url,
355     repo_absolute_path: None,
356 })
357 }
358
359 /// Lists all files to be included in the PDF.
360 ///
361 /// In git mode: uses `git ls-files` (working tree) or `git ls-tree` (specific
362 /// branch/commit). In plain-directory mode: recursively walks the filesystem.
363 ///
364 /// # Errors
365 ///
366 /// Returns an error if the git command or directory walk fails.
367 pub async fn list_tracked_files(
368     repo_path: &Path,
369     config: &Config,
370     is_git: bool,
371     scope: Option<&Path>,
372 ) -> anyhow::Result<Vec<PathBuf>> {
373     if !is_git {
374         return walk_files_async(repo_path.to_path_buf()).await;
375     }
376
377     let scope_str = scope.and_then(|p| p.to_str());
378     let output = match (&config.commit, &config.branch) {
379         (Some(commit), _) => match scope_str {
380             Some(s) => {
381                 run_git(
382                     repo_path,
383                     &["ls-tree", "-r", "--name-only", commit, "--", s],
384                 )
385                 .await?
386             }
387             None => run_git(repo_path, &["ls-tree", "-r", "--name-only", commit]).await?,
388         },
389         (_, Some(branch)) => match scope_str {

```

```

390     Some(s) => {
391         run_git(
392             repo_path,
393             &["ls-tree", "-r", "--name-only", branch, "--", s],
394         )
395         .await?
396     }
397     None => run_git(repo_path, &["ls-tree", "-r", "--name-only", branch]).await?,
398 },
399 _ => match scope_str {
400     Some(s) => run_git(repo_path, &["ls-files", "--", s]).await?,
401     None => run_git(repo_path, &["ls-files"]).await?,
402 },
403 };
404
405 Ok(output
406     .lines()
407     .filter(|l| !l.is_empty())
408     .map(PathBuf::from)
409     .collect()
410 }
411
412 /// Returns a map of file path → last modified date (YYYY-MM-DD).
413 /// In git mode: parsed from `git log`. In directory mode: from filesystem mtime.
414 pub async fn file_last_modified_dates(
415     repo_path: &Path,
416     config: &Config,
417     is_git: bool,
418     scope: Option<&Path>,
419 ) -> anyhow::Result<HashMap<PathBuf, String>> {
420     if !is_git {
421         return walk_dates_async(repo_path.to_path_buf()).await;
422     }
423
424     let rev = match (&config.commit, &config.branch) {
425         (Some(c), _) => c.clone(),
426         (_, Some(b)) => b.clone(),
427         _ => "HEAD".to_string(),
428     };
429
430     let scope_str = scope.and_then(|p| p.to_str());
431     let output = match scope_str {
432         Some(s) => {
433             run_git(
434                 repo_path,
435                 &["log", "--format=COMMIT:%ci", "--name-only", &rev, "--", s],
436             )
437             .await?
438         }
439         None => {
440             run_git(
441                 repo_path,
442                 &["log", "--format=COMMIT:%ci", "--name-only", &rev],
443             )
444             .await?
445         }
446     };
447
448     let mut map = HashMap::new();
449     let mut current_date = String::new();
450
451     output.lines().for_each(|line| {
452         if let Some(date_str) = line.strip_prefix("COMMIT:") {
453             current_date = date_str.chars().take(10).collect();
454         } else if !line.is_empty() && !current_date.is_empty() {
455             map.entry(PathBuf::from(line))
456                 .or_insert_with(|| current_date.clone());
457         }
458     });
459
460     Ok(map)
461 }
462
463 /// Returns the last-modified date (YYYY-MM-DD) for a single file.
464 /// In git mode: from `git log`. In plain mode: from filesystem mtime.
465 pub async fn file_last_modified(root: &Path, file: &Path, config: &Config, is_git: bool) -> String {
466     if is_git {
467         let rev = config

```

```
468         .commit
469         .as_deref()
470         .or(config.branch.as_deref())
471         .unwrap_or("HEAD");
472     let file_str = file.to_string_lossy();
473     run_git(
474         root,
475         &["log", "-1", "--format=%ci", rev, "--", file_str.as_ref()],
476     )
477     .await
478     .ok()
479     .map(|s| s.trim().chars().take(10).collect())
480     .unwrap_or_default()
481 } else {
482     tokio::fs::metadata(root.join(file))
483     .await
484     .ok()
485     .and_then(|m| m.modified().ok())
486     .map(|t| {
487         let secs = t.duration_since(UNIX_EPOCH).unwrap_or_default().as_secs();
488         let (y, m, d) = unix_secs_to_ymd(secs);
489         format!("{y:04}-{m:02}-{d:02}")
490     })
491     .unwrap_or_default()
492 }
493 }
494
495 /// Reads the content of a single file, using `git show` for a specific revision or plain I/O otherwise.
496 pub async fn read_file_content(
497     repo_path: &Path,
498     file_path: &Path,
499     config: &Config,
500 ) -> anyhow::Result<String> {
501     let rev = config.commit.as_deref().or(config.branch.as_deref());
502     match rev {
503         Some(rev) => {
504             let spec = format!("{rev}:{}", file_path.display());
505             run_git(repo_path, &["show", &spec]).await
506         }
507         None => tokio::fs::read_to_string(repo_path.join(file_path))
508             .await
509             .map_err(Into::into),
510     }
511 }
512
513 // — Private helpers for plain-directory mode —————
514
515 /// Converts Unix timestamp (seconds since epoch) to (year, month, day).
516 /// Uses Howard Hinnant's date algorithm.
517 fn unix_secs_to_ymd(secs: u64) -> (u32, u32, u32) {
518     let z = (secs / 86400) as i64 + 719_468;
519     let era = if z >= 0 { z } else { z - 146_096 } / 146_097;
520     let doe = (z - era * 146_097) as u32;
521     let yoe = (doe - doe / 1460 + doe / 36524 - doe / 146096) / 365;
522     let y = yoe as i64 + era * 400;
523     let doy = doe - (365 * yoe + yoe / 4 - yoe / 100);
524     let mp = (5 * doy + 2) / 153;
525     let d = doy - (153 * mp + 2) / 5 + 1;
526     let m = if mp < 10 { mp + 3 } else { mp - 9 };
527     let y = if m <= 2 { y + 1 } else { y };
528     (y as u32, m, d)
529 }
530
531 /// Recursive async walk returning all file paths relative to `root`.
532 /// Each directory immediately spawns tasks for its subdirectories - no
533 /// level-by-level BFS barriers, maximum concurrency throughout the tree.
534 fn walk_files_inner(
535     root: Arc<PathBuf>,
536     dir: PathBuf,
537 ) -> Pin<Box<dyn Future<Output = anyhow::Result<Vec<PathBuf>>> + Send>> {
538     Box::pin(async move {
539         let mut rd = tokio::fs::read_dir(&dir).await?;
540         let mut files: Vec<PathBuf> = Vec::new();
541         let mut set: tokio::task::JoinSet<anyhow::Result<Vec<PathBuf>>> =
542             tokio::task::JoinSet::new();
543
544         while let Some(entry) = rd.next_entry().await? {
545             let ft = entry.file_type().await?;
```

```

546         if ft.is_dir() {
547             set.spawn(walk_files_inner(Arc::clone(&root), entry.path()));
548         } else if ft.is_file()
549             && let Ok(rel) = entry.path().strip_prefix(root.as_ref())
550         {
551             files.push(rel.to_path_buf());
552         }
553     }
554
555     set.join_all()
556         .await
557         .into_iter()
558         .try_for_each(|res| res.map(|sub| files.extend(sub)))?;
559
560     Ok(files)
561 })
562 }
563
564 async fn walk_files_async(root: PathBuf) -> anyhow::Result<Vec<PathBuf>> {
565     walk_files_inner(Arc::new(root.clone()), root).await
566 }
567
568 /// Walk the tree (via `walk_files_async`) then fetch all file mtimes concurrently.
569 async fn walk_dates_async(root: PathBuf) -> anyhow::Result<HashMap<PathBuf, String>> {
570     let files = walk_files_async(root.clone()).await?;
571     let mut set: tokio::task::JoinSet<Option<(PathBuf, String)>> = tokio::task::JoinSet::new();
572
573     files.into_iter().for_each(|rel| {
574         let abs = root.join(&rel);
575         set.spawn(async move {
576             let date = tokio::fs::metadata(&abs)
577                 .await
578                 .ok()
579                 .and_then(|m| m.modified().ok())
580                 .map(|t| {
581                     let secs = t.duration_since(UNIX_EPOCH).unwrap_or_default().as_secs();
582                     let (y, m, d) = unix_secs_to_ymd(secs);
583                     format!("{y:04}-{m:02}-{d:02}")
584                 })?;
585             Some((rel, date))
586         });
587     });
588
589     Ok(set.join_all().await.into_iter().flatten().collect())
590 }
591
592 /// Returns the filesystem owner username and group name for `path`.
593 ///
594 /// Tries GNU `stat -c "%U\n%G"` (Linux/coreutils) then BSD `stat -f "%Su\n%Sg"` (macOS).
595 /// Returns `(None, None)` if both fail or the path is inaccessible.
596 pub async fn fs_owner_group(path: &Path) -> (Option<String>, Option<String>) {
597     for args in [
598         &["-c", "%U\n%G"][..], // GNU stat (Linux)
599         &["-f", "%Su\n%Sg"][..], // BSD stat (macOS)
600     ] {
601         if let Ok(out) = Command::new("stat").args(args).arg(path).output().await {
602             if out.status.success() {
603                 let text = String::from_utf8_lossy(&out.stdout);
604                 let mut lines = text.trim().lines();
605                 return (
606                     lines.next().filter(|s| !s.is_empty()).map(str::to_string),
607                     lines.next().filter(|s| !s.is_empty()).map(str::to_string),
608                 );
609             }
610         }
611     }
612     (None, None)
613 }
614
615 /// Formats a byte count as a human-readable string using binary prefixes.
616 fn format_bytes(bytes: u64) -> String {
617     if bytes < 1_024 {
618         format!("{bytes} B")
619     } else if bytes < 1_048_576 {
620         format!("{:.1} KB", bytes as f64 / 1_024.0)
621     } else if bytes < 1_073_741_824 {
622         format!("{:.1} MB", bytes as f64 / 1_048_576.0)
623     } else {

```

```
624     format!("{:.1} GB", bytes as f64 / 1_073_741_824.0)
625 }
626 }
627
628 /// Recursive half of `fs_dir_size`: sums the byte sizes of all regular files under `dir`.
629 fn sum_dir_bytes(dir: PathBuf) -> Pin<Box<dyn Future<Output = u64> + Send>> {
630     Box::pin(async move {
631         let mut rd = match tokio::fs::read_dir(&dir).await {
632             Ok(rd) => rd,
633             Err(_) => return 0,
634         };
635         let mut total = 0u64;
636         let mut set: tokio::task::JoinSet<u64> = tokio::task::JoinSet::new();
637         while let Ok(Some(entry)) = rd.next_entry().await {
638             let Ok(ft) = entry.file_type().await else {
639                 continue;
640             };
641             if ft.is_dir() {
642                 set.spawn(sum_dir_bytes(entry.path()));
643             } else if ft.is_file() {
644                 if let Ok(m) = entry.metadata().await {
645                     total += m.len();
646                 }
647             }
648         }
649         total += set.join_all().await.into_iter().sum::<u64>();
650     total
651 })
652 }
653
654 /// Returns the total size of all files under `path` as a human-readable string (e.g. `4.2 MB`).
655 ///
656 /// Recursively sums actual file content bytes (not disk-block allocations) using
657 /// async I/O – no subprocess, no platform-specific tools, consistent formatting.
658 /// Returns an empty string if `path` is inaccessible or contains no files.
659 pub async fn fs_dir_size(path: &Path) -> String {
660     let total = sum_dir_bytes(path.to_path_buf()).await;
661     if total == 0 {
662         String::new()
663     } else {
664         format_bytes(total)
665     }
666 }
667
668 /// Returns the total size of all git-tracked files as a human-readable string (e.g. `3.8 MB`).
669 ///
670 /// Uses `git ls-tree -r -l` to sum blob sizes from the object database – this
671 /// reflects actual tracked content without `.git/` overhead.
672 /// Falls back to an empty string on failure.
673 pub async fn git_tracked_size(repo_path: &Path, config: &Config) -> String {
674     let rev = config
675         .commit
676         .as_deref()
677         .or(config.branch.as_deref())
678         .unwrap_or("HEAD");
679     let output = run_git(repo_path, &["ls-tree", "-r", "-l", rev])
680         .await
681         .unwrap_or_default();
682     let total_bytes: u64 = output
683         .lines()
684         .filter_map(|line| line.split_whitespace().nth(3)?.parse::<u64>().ok())
685         .sum();
686     if total_bytes == 0 {
687         return String::new();
688     }
689     format_bytes(total_bytes)
690 }
691
692 /// Normalizes a git remote URL to an `https://` URL.
693 ///
694 /// Handles SCP-style (`git@github.com:user/repo`) and `ssh://` URLs, converting
695 /// them to their `https://` equivalents so they can be used in clickable links.
696 /// Plain `https://` or `http://` URLs are returned unchanged.
697 fn normalize_to_https(url: &str) -> String {
698     if url.starts_with("https://") || url.starts_with("http://") {
699         return url.to_string();
700     } else if let Some(rest) = url.strip_prefix("git@") {
701         // SCP-style: git@github.com:user/repo.git

```

```

702     if let Some(colon_pos) = rest.find(':') {
703         return format!("https://{}/{}", &rest[..colon_pos], &rest[colon_pos + 1..]);
704     }
705 } else if let Some(rest) = url.strip_prefix("ssh://git@") {
706     return format!("https://{rest}");
707 } else if let Some(rest) = url.strip_prefix("ssh://") {
708     return format!("https://{rest}");
709 }
710 url.to_string()
711 }
712
713 /// Fetches all tag refs from the remote without downloading full history.
714 ///
715 /// Needed after a `--depth=1` clone, which only fetches the tag (if any)
716 /// pointing at the cloned commit - other tags are absent until this runs.
717 pub async fn fetch_tags(repo_path: &Path) -> anyhow::Result<()> {
718     let status = Command::new("git")
719         .args(["fetch", "--tags", "--depth=1"])
720         .current_dir(repo_path)
721         .status()
722         .await
723         .map_err(|e| anyhow::anyhow!("failed to run git: {e}"))?;
724     if !status.success() {
725         bail!("git fetch --tags failed");
726     }
727     Ok(())
728 }
729
730 /// Lists all version tags in a repository, sorted newest first.
731 ///
732 /// Uses `git tag --list --sort=-version:refname` which sorts by semver-aware
733 /// descending order so `v1.10.0` sorts before `v1.9.0`.
734 /// Returns an empty Vec if there are no tags or the path is not a git repo.
735 pub async fn list_repo_tags(repo_path: &Path) -> Vec<String> {
736     run_git(repo_path, &["tag", "--list", "--sort=-version:refname"])
737         .await
738         .unwrap_or_default()
739         .lines()
740         .filter(|l| !l.is_empty())
741         .map(str::to_string)
742         .collect()
743 }
744
745 /// Returns the remote URL for `origin`, if one is configured.
746 ///
747 /// Runs `git remote get-url origin` - if the repo has no remote or the command
748 /// fails, returns `None`. SCP-style and ssh:// URLs are normalized to https://.
749 pub async fn git_remote_url(repo_path: &Path) -> Option<String> {
750     run_git(repo_path, &["remote", "get-url", "origin"])
751         .await
752         .ok()
753         .map(|s| normalize_to_https(s.trim()))
754         .filter(|s| !s.is_empty())
755 }
756
757 #[cfg(test)]
758 mod tests {
759     use super::*;
760
761     #[test]
762     fn normalize_https_passthrough() {
763         assert_eq!(
764             normalize_to_https("https://github.com/user/repo.git"),
765             "https://github.com/user/repo.git"
766         );
767         assert_eq!(
768             normalize_to_https("http://example.com/repo"),
769             "http://example.com/repo"
770         );
771     }
772
773     #[test]
774     fn normalize_scp_style() {
775         assert_eq!(
776             normalize_to_https("git@github.com:user/repo.git"),
777             "https://github.com/user/repo.git"
778         );
779         assert_eq!(

```

```
780         normalize_to_https("git@gitlab.com:org/project"),
781         "https://gitlab.com/org/project"
782     );
783 }
784
785 #[test]
786 fn normalize_ssh_git_at_style() {
787     assert_eq!(
788         normalize_to_https("ssh://git@github.com/user/repo.git"),
789         "https://github.com/user/repo.git"
790     );
791 }
792
793 #[test]
794 fn normalize_ssh_style() {
795     assert_eq!(
796         normalize_to_https("ssh://github.com/user/repo.git"),
797         "https://github.com/user/repo.git"
798     );
799 }
800
801 #[test]
802 fn is_remote_url_https() {
803     assert!(is_remote_url("https://github.com/user/repo"));
804     assert!(is_remote_url("https://github.com/user/repo.git"));
805     assert!(is_remote_url("http://example.com/repo.git"));
806 }
807
808 #[test]
809 fn is_remote_url_git_schemes() {
810     assert!(is_remote_url("git://github.com/user/repo.git"));
811     assert!(is_remote_url("ssh://git@github.com/user/repo.git"));
812 }
813
814 #[test]
815 fn is_remote_url_scp_style() {
816     assert!(is_remote_url("git@github.com:user/repo.git"));
817     assert!(is_remote_url("git@gitlab.com:org/repo"));
818 }
819
820 #[test]
821 fn is_remote_url_rejects_local() {
822     assert!(is_remote_url("."));
823     assert!(is_remote_url("/home/user/repo"));
824     assert!(is_remote_url("relative/path"));
825     assert!(is_remote_url("src/main.rs"));
826 }
827
828 #[test]
829 fn repo_name_from_url_https() {
830     assert_eq!(
831         repo_name_from_url("https://github.com/user/repo.git"),
832         "repo"
833     );
834     assert_eq!(repo_name_from_url("https://github.com/user/repo"), "repo");
835 }
836
837 #[test]
838 fn repo_name_from_url_scp() {
839     assert_eq!(repo_name_from_url("git@github.com:user/repo.git"), "repo");
840     assert_eq!(
841         repo_name_from_url("git@gitlab.com:org/myproject"),
842         "myproject"
843     );
844 }
845
846 #[tokio::test]
847 async fn temp_clone_dir_creates_and_cleans_up() {
848     let path = {
849         let t = TempCloneDir::for_url("https://example.com/repo", None, None)
850             .await
851             .unwrap();
852         let p = t.path().to_path_buf();
853         assert!(p.exists());
854         p
855     };
856     assert!(!path.exists());
857 }
```



## src/github.rs

435 LOC · 14.1 KB · 2026-03-14

```

1  ///! GitHub REST API v3 client.
2  ///!
3  ///! All functions operate on public data and work without authentication.
4  ///! Set `GITHUB_TOKEN` in the environment for higher rate limits (5 000/hr vs 60/hr)
5  ///! and access to private repositories.
6
7  use anyhow::{Context, bail};
8  use serde::Deserialize;
9
10 const API_BASE: &str = "https://api.github.com";
11 const VERSION: &str = env!("CARGO_PKG_VERSION");
12
13 // — Response types —
14
15 ///! GitHub user public profile returned by `GET /users/{username}`.
16 #[allow(missing_docs)]
17 #[derive(Debug, Deserialize)]
18 pub struct GitHubUser {
19     pub login: String,
20     pub name: Option<String>,
21     pub bio: Option<String>,
22     pub location: Option<String>,
23     pub company: Option<String>,
24     pub blog: Option<String>,
25     pub email: Option<String>,
26     pub public_repos: u64,
27     pub followers: u64,
28     pub following: u64,
29     pub created_at: String,
30     pub html_url: String,
31 }
32
33 ///! A GitHub repository as returned by the repos and search APIs.
34 #[allow(missing_docs)]
35 #[derive(Debug, Deserialize, Clone)]
36 pub struct GitHubRepo {
37     pub name: String,
38     pub full_name: String,
39     pub html_url: String,
40     pub description: Option<String>,
41     pub language: Option<String>,
42     pub stargazers_count: u64,
43     pub forks_count: u64,
44     pub pushed_at: Option<String>,
45     pub updated_at: Option<String>,
46     pub fork: bool,
47     #[serde(default)]
48     pub open_issues_count: u64,
49     #[serde(default)]
50     pub size: u64, // in KB
51     #[serde(default)]
52     pub created_at: Option<String>,
53 }
54
55 ///! A public GitHub event as returned by `GET /users/{username}/events/public`.
56 #[allow(missing_docs)]
57 #[derive(Debug, Deserialize)]
58 pub struct GitHubEvent {
59     #[serde(rename = "type")]
60     pub kind: String,
61     pub repo: EventRepo,
62     pub payload: serde_json::Value,
63     pub created_at: String,
64 }
65
66 ///! The repository reference embedded in a GitHub event.
67 #[allow(missing_docs)]
68 #[derive(Debug, Deserialize)]
69 pub struct EventRepo {
70     pub name: String,
71 }
72
73 ///! A single commit with its file patches, as returned by `GET /repos/{owner}/{repo}/commits/{sha}`.
74 #[allow(missing_docs)]
75 #[derive(Debug, Deserialize)]
76 pub struct CommitDetail {
77     pub sha: String,

```

```
78     pub html_url: String,
79     pub commit: CommitInfo,
80     #[serde(default)]
81     pub files: Vec<CommitFile>,
82 }
83
84 /// Commit metadata (message and author) embedded in a `CommitDetail`.
85 #[allow(missing_docs)]
86 #[derive(Debug, Deserialize)]
87 pub struct CommitInfo {
88     pub message: String,
89     pub author: CommitAuthor,
90 }
91
92 /// Author name and date embedded in a `CommitInfo`.
93 #[allow(missing_docs)]
94 #[derive(Debug, Deserialize)]
95 pub struct CommitAuthor {
96     pub name: String,
97     pub date: String,
98 }
99
100 /// A single changed file within a commit, including optional unified diff patch.
101 #[allow(missing_docs)]
102 #[derive(Debug, Deserialize)]
103 pub struct CommitFile {
104     pub filename: String,
105     pub status: String,
106     pub additions: u64,
107     pub deletions: u64,
108     pub patch: Option<String>,
109 }
110
111 // — Client helpers
112
113 pub(crate) fn build_client() -> anyhow::Result<request::Client> {
114     request::Client::builder()
115         .user_agent(format!("gitprint/{VERSION}"))
116         .build()
117         .context("failed to build HTTP client")
118 }
119
120 fn auth_header(token: Option<&str>) -> Option<String> {
121     token.map(|t| format!("Bearer {t}"))
122 }
123
124 pub(crate) async fn get_json<T: for<'de> Deserialize<'de>>>(
125     client: &request::Client,
126     url: &str,
127     token: Option<&str>,
128 ) -> anyhow::Result<T> {
129     let mut req = client
130         .get(url)
131         .header("Accept", "application/vnd.github+json");
132     if let Some(auth) = auth_header(token) {
133         req = req.header("Authorization", auth);
134     }
135     let resp = req.send().await.with_context(|| format!("GET {url}"))?;
136     let status = resp.status();
137     if status == request::StatusCode::NOT_FOUND {
138         bail!("not found: {url}");
139     }
140     if status == request::StatusCode::FORBIDDEN || status == request::StatusCode::TOO_MANY_REQUESTS
141     {
142         bail!(
143             "GitHub API rate limit exceeded. Set GITHUB_TOKEN to increase limits:\n \
144             export GITHUB_TOKEN=ghp_your_token_here"
145         );
146     }
147     if !status.is_success() {
148         bail!("GitHub API error {status}: {url}");
149     }
150     resp.json:::<T>()
151         .await
152         .with_context(|| format!("parsing response from {url}"))
153 }
154
155 // — Public API functions
```

```
156
157 // Fetch a user's public profile.
158 pub async fn get_user(username: &str, token: Option<&str>) -> anyhow::Result<GitHubUser> {
159     let client = build_client()?;
160     let url = format!("{API_BASE}/users/{username}");
161     get_json::<GitHubUser>(&client, &url, token)
162         .await
163         .with_context(|| format!("fetching user '{username}'"))
164 }
165
166 // Wrapper for the GitHub search/repositories response.
167 #[derive(Debug, Deserialize)]
168 struct SearchReposResponse {
169     items: Vec<GitHubRepo>,
170 }
171
172 // Fetch a user's top starred repositories via the Search API.
173 //
174 // Uses `/search/repositories` because `/users/{u}/repos` does not support `sort=stars`.
175 pub async fn get_user_starred_repos(
176     username: &str,
177     limit: usize,
178     token: Option<&str>,
179 ) -> anyhow::Result<Vec<GitHubRepo>> {
180     let client = build_client()?;
181     let per_page = limit.min(100);
182     let url = format!(
183         "{API_BASE}/search/repositories?q=user:{username}+fork:false&sort=stars&order=desc&per_page={per_page}"
184     );
185     get_json::<SearchReposResponse>(&client, &url, token)
186         .await
187         .map(|r| r.items)
188         .with_context(|| format!("fetching starred repos for '{username}'"))
189 }
190
191 // Fetch a user's own repositories sorted by `sort` (`pushed` or `updated`).
192 //
193 // `limit` is capped at 100 (GitHub's maximum per-page).
194 // Only returns repos the user owns directly (`type=owner`).
195 pub async fn get_user_repos(
196     username: &str,
197     sort: &str,
198     limit: usize,
199     token: Option<&str>,
200 ) -> anyhow::Result<Vec<GitHubRepo>> {
201     let client = build_client()?;
202     let per_page = limit.min(100);
203     let url = format!(
204         "{API_BASE}/users/{username}/repos?type=owner&sort={sort}&direction=desc&per_page={per_page}"
205     );
206     get_json::<Vec<GitHubRepo>>(&client, &url, token)
207         .await
208         .with_context(|| format!("fetching repos for '{username}' (sort={sort})"))
209 }
210
211 // Fetch a user's recent public events (max 100, GitHub returns up to 90 days).
212 pub async fn get_user_events(
213     username: &str,
214     limit: usize,
215     token: Option<&str>,
216 ) -> anyhow::Result<Vec<GitHubEvent>> {
217     let client = build_client()?;
218     let per_page = limit.min(100);
219     let url = format!("{API_BASE}/users/{username}/events/public?per_page={per_page}");
220     get_json::<Vec<GitHubEvent>>(&client, &url, token)
221         .await
222         .with_context(|| format!("fetching events for '{username}'"))
223 }
224
225 // Response envelope for the commits search endpoint.
226 #[derive(Deserialize)]
227 struct CommitSearchResponse {
228     items: Vec<CommitSearchItem>,
229 }
230
231 #[derive(Deserialize)]
232 struct CommitSearchItem {
233     sha: String,
```

```

234     repository: CommitSearchRepo,
235     commit: CommitSearchMeta,
236 }
237
238 #[derive(Deserialize)]
239 struct CommitSearchRepo {
240     full_name: String,
241 }
242
243 #[derive(Deserialize)]
244 struct CommitSearchMeta {
245     message: String,
246 }
247
248 /// Search for the `limit` most recent public commits authored by `username` across all repos.
249 ///
250 /// Uses `GET /search/commits?q=author:{username}` (stable since GitHub API v3 2022+).
251 /// Returns `(owner/repo, sha, first-line-of-message)` tuples, newest first.
252 /// Returns an empty Vec on error so the caller can degrade gracefully.
253 pub async fn search_user_commits(
254     username: &str,
255     limit: usize,
256     token: Option<&str>,
257 ) -> anyhow::Result<Vec<(String, String, String)>> {
258     let client = build_client()?;
259     let per_page = limit.min(100);
260     let url = format!(
261         "{API_BASE}/search/commits?q=author:{username}&sort=committer-date&order=desc&per_page={per_page}"
262     );
263     get_json::<CommitSearchResponse>(&client, &url, token)
264         .await
265         .map(|r| {
266             r.items
267                 .into_iter()
268                 .map(|item| {
269                     let msg = item
270                         .commit
271                         .message
272                         .lines()
273                         .next()
274                         .unwrap_or(&item.commit.message)
275                         .to_string();
276                     (item.repository.full_name, item.sha, msg)
277                 })
278                 .collect()
279         })
280         .with_context(|| format!("searching commits by '{username}'"))
281 }
282
283 /// Fetch a single commit with its file patches.
284 pub async fn get_commit_detail(
285     owner_repo: &str,
286     sha: &str,
287     token: Option<&str>,
288 ) -> anyhow::Result<CommitDetail> {
289     let client = build_client()?;
290     let url = format!("{API_BASE}/repos/{owner_repo}/commits/{sha}");
291     get_json::<CommitDetail>(&client, &url, token)
292         .await
293         .with_context(|| format!("fetching commit {sha} in {owner_repo}"))
294 }
295
296 #[cfg(test)]
297 mod tests {
298     use super::*;
299     use httpmock::prelude::*;
300
301     #[test]
302     fn auth_header_some() {
303         assert_eq!(auth_header(Some("tok")), Some("Bearer tok".to_string()));
304     }
305
306     #[test]
307     fn auth_header_none() {
308         assert_eq!(auth_header(None), None);
309     }
310
311     #[tokio::test]

```

```
312 async fn parses_user_response() -> anyhow::Result<()> {
313     let server = MockServer::start();
314     server.mock(|when, then| {
315         when.method(GET).path("/users/alice");
316         then.status(200).json_body(serde_json::json!({
317             "login": "alice", "name": "Alice", "bio": null, "location": null,
318             "company": null, "blog": null, "email": null, "public_repos": 10,
319             "followers": 42, "following": 5, "created_at": "2020-01-01T00:00:00Z",
320             "html_url": "https://github.com/alice"
321         })));
322     });
323
324     let client = build_client()?;
325     let user: GitHubUser =
326         get_json(&client, &format!("{}/users/alice", server.base_url()), None).await?;
327     assert_eq!(user.login, "alice");
328     assert_eq!(user.public_repos, 10);
329     assert_eq!(user.followers, 42);
330     Ok(())
331 }
332
333 #[tokio::test]
334 async fn parses_repo_list_response() -> anyhow::Result<()> {
335     let server = MockServer::start();
336     server.mock(|when, then| {
337         when.method(GET).path("/users/alice/repos");
338         then.status(200).json_body(serde_json::json!([
339             {
340                 "name": "myrepo", "full_name": "alice/myrepo",
341                 "html_url": "https://github.com/alice/myrepo", "description": null,
342                 "language": "Rust", "stargazers_count": 7, "forks_count": 1,
343                 "pushed_at": "2024-03-01T00:00:00Z", "updated_at": "2024-03-01T00:00:00Z",
344                 "fork": false
345             }
346         ])));
347     });
348
349     let client = build_client()?;
350     let repos: Vec<GitHubRepo> = get_json(
351         &client,
352         &format!("{}/users/alice/repos", server.base_url()),
353         None,
354     )
355     .await?;
356     assert_eq!(repos.len(), 1);
357     assert_eq!(repos[0].name, "myrepo");
358     assert_eq!(repos[0].stargazers_count, 7);
359     Ok(())
360 }
361
362 #[tokio::test]
363 async fn parses_event_list_response() -> anyhow::Result<()> {
364     let server = MockServer::start();
365     server.mock(|when, then| {
366         when.method(GET).path("/users/alice/events/public");
367         then.status(200).json_body(serde_json::json!([
368             {
369                 "type": "PushEvent",
370                 "repo": { "name": "alice/myrepo" },
371                 "payload": { "ref": "refs/heads/main", "commits": [] },
372                 "created_at": "2024-03-01T12:00:00Z"
373             }
374         ])));
375     });
376
377     let client = build_client()?;
378     let events: Vec<GitHubEvent> = get_json(
379         &client,
380         &format!("{}/users/alice/events/public", server.base_url()),
381         None,
382     )
383     .await?;
384     assert_eq!(events.len(), 1);
385     assert_eq!(events[0].kind, "PushEvent");
386     assert_eq!(events[0].repo.name, "alice/myrepo");
387     Ok(())
388 }
389
390 #[tokio::test]
391 async fn parses_commit_detail_response() -> anyhow::Result<()> {
392     let server = MockServer::start();
393     let sha = "abc1234abc1234abc1234abc1234abc1234abc1234";
394 }
```

```
390 server.mock(|when, then| {
391     when.method(GET)
392         .path(format!("/repos/alice/myrepo/commits/{sha}"));
393     then.status(200).json_body(serde_json::json!({
394         "sha": sha,
395         "html_url": "https://github.com/alice/myrepo/commit/abc1234",
396         "commit": {
397             "message": "fix: handle edge case",
398             "author": { "name": "Alice", "date": "2024-03-01T12:00:00Z" }
399         },
400         "files": [{
401             "filename": "src/lib.rs", "status": "modified",
402             "additions": 5, "deletions": 2, "patch": "+added line\n-removed line"
403         }]
404     }));
405 });
406
407 let client = build_client()?;
408 let detail: CommitDetail = get_json(
409     &client,
410     &format!("{}/repos/alice/myrepo/commits/{sha}", server.base_url()),
411     None,
412 )
413 .await?;
414 assert_eq!(detail.sha, sha);
415 assert_eq!(detail.commit.message, "fix: handle edge case");
416 assert_eq!(detail.files[0].additions, 5);
417 Ok(())
418 }
419
420 #[tokio::test]
421 async fn rate_limit_error_is_surfaced() {
422     let server = MockServer::start();
423     server.mock(|when, then| {
424         when.method(GET).path("/users/alice");
425         then.status(403);
426     });
427
428     let client = build_client().unwrap();
429     let err =
430         get_json:::<GitHubUser>(&client, &format!("{}/users/alice", server.base_url()), None)
431         .await
432         .unwrap_err();
433     assert!(err.to_string().contains("rate limit"), "got: {err}");
434 }
435 }
```

## src/highlight.rs

```

1 use std::path::Path;
2
3 use syntect::easy::HighlightLines;
4 use syntect::highlighting::{FontStyle, ThemeSet};
5 use syntect::parsing::SyntaxSet;
6
7 use crate::types::{HighlightedLine, HighlightedToken, RgbColor};
8
9 /// Syntax highlighter backed by the bundled syntect theme and syntax sets.
10 pub struct Highlighter {
11     syntax_set: SyntaxSet,
12     theme: syntect::highlighting::Theme,
13 }
14
15 impl Highlighter {
16     /// Creates a new `Highlighter` using the named syntect theme.
17     ///
18     /// Theme names are the keys returned by `list_themes`. Pass `InspiredGitHub` for
19     /// the default light theme.
20     ///
21     /// # Errors
22     ///
23     /// Returns an error if `theme_name` is not found in the bundled theme set.
24     ///
25     /// # Examples
26     ///
27     /// ```
28     /// use gitprint::highlight::Highlighter;
29     ///
30     /// let hl = Highlighter::new("InspiredGitHub").unwrap();
31     ///
32     /// let err = Highlighter::new("no-such-theme").err().unwrap();
33     /// assert!(err.to_string().contains("no-such-theme"));
34     /// ```
35     pub fn new(theme_name: &str) -> anyhow::Result<Self> {
36         let syntax_set = SyntaxSet::load_defaults_newlines();
37         let theme_set = ThemeSet::load_defaults();
38
39         let theme = theme_set.themes.get(theme_name).cloned().ok_or_else(|| {
40             anyhow::anyhow!(
41                 "theme not found: {theme_name} (use --list-themes to see available themes)"
42             )
43         })?;
44
45         Ok(Self { syntax_set, theme })
46     }
47
48     /// Returns a lazy iterator that yields one `HighlightedLine` at a time.
49     ///
50     /// Syntax is detected from the file extension of `path`; unknown extensions fall
51     /// back to plain text. Line numbers start at 1.
52     ///
53     /// # Examples
54     ///
55     /// ```
56     /// use gitprint::highlight::Highlighter;
57     /// use std::path::Path;
58     ///
59     /// let hl = Highlighter::new("InspiredGitHub").unwrap();
60     /// let lines: Vec<_> = hl.highlight_lines("fn main() {}", Path::new("main.rs")).collect();
61     ///
62     /// assert_eq!(lines.len(), 1);
63     /// assert_eq!(lines[0].line_number, 1);
64     /// assert!(lines[0].tokens.is_empty());
65     /// ```
66     pub fn highlight_lines<'a>(
67         &'a self,
68         content: &'a str,
69         path: &Path,
70     ) -> impl Iterator<Item = HighlightedLine> + 'a {
71         let syntax = self
72             .syntax_set
73             .find_syntax_for_file(path)
74             .ok()
75             .flatten()
76             .unwrap_or_else(|| self.syntax_set.find_syntax_plain_text());
77

```

```

78     let mut h = HighlightLines::new(syntax, &self.theme);
79     let mut lines = content.lines().enumerate();
80
81     std::iter::from_fn(move || {
82         let (i, line_text) = lines.next()?;
83
84         let tokens = h
85             .highlight_line(line_text, &self.syntax_set)
86             .unwrap_or_default()
87             .into_iter()
88             .map(|(style, text)| HighlightedToken {
89                 text: text.to_string(),
90                 color: RgbColor {
91                     r: style.foreground.r,
92                     g: style.foreground.g,
93                     b: style.foreground.b,
94                 },
95                 bold: style.font_style.contains(FontStyle::BOLD),
96                 italic: style.font_style.contains(FontStyle::ITALIC),
97             })
98             .collect();
99
100        Some(HighlightedLine {
101            line_number: i + 1,
102            tokens,
103        })
104    })
105 }
106 }
107
108 /// Returns all available theme names in sorted order.
109 ///
110 /// # Examples
111 ///
112 /// ```
113 /// use gitprint::highlight::list_themes;
114 ///
115 /// let themes = list_themes();
116 /// assert!(themes.contains(&"InspiredGitHub".to_string()));
117 /// assert!(themes.windows(2).all(|w| w[0] <= w[1])); // sorted
118 /// ```
119 pub fn list_themes() -> Vec<String> {
120     let mut themes: Vec<_> = ThemeSet::load_defaults().themes.into_keys().collect();
121     themes.sort();
122     themes
123 }
124
125 #[cfg(test)]
126 mod tests {
127     use super::*;
128
129     #[test]
130     fn new_with_valid_theme() {
131         assert!(Highlighter::new("InspiredGitHub").is_ok());
132     }
133
134     #[test]
135     fn new_with_another_valid_theme() {
136         assert!(Highlighter::new("base16-ocean.dark").is_ok());
137     }
138
139     #[test]
140     fn new_with_invalid_theme() {
141         let result = Highlighter::new("NonExistentTheme");
142         assert!(result.is_err());
143         assert!(
144             result
145                 .err()
146                 .unwrap()
147                 .to_string()
148                 .contains("NonExistentTheme")
149         );
150     }
151
152     #[test]
153     fn highlight_lines_produces_output() {
154         let h = Highlighter::new("InspiredGitHub").unwrap();
155         let lines: Vec<_> = h

```

```

156         .highlight_lines("fn main() {}", Path::new("test.rs"))
157         .collect();
158     assert_eq!(lines.len(), 1);
159     assert_eq!(lines[0].line_number, 1);
160     assert!(lines[0].tokens.is_empty());
161 }
162
163 #[test]
164 fn highlight_lines_multiline() {
165     let h = Highlighter::new("InspiredGitHub").unwrap();
166     let content = "line1\nline2\nline3";
167     let lines: Vec<_> = h.highlight_lines(content, Path::new("test.txt")).collect();
168     assert_eq!(lines.len(), 3);
169     assert_eq!(lines[0].line_number, 1);
170     assert_eq!(lines[1].line_number, 2);
171     assert_eq!(lines[2].line_number, 3);
172 }
173
174 #[test]
175 fn highlight_lines_preserves_text() {
176     let h = Highlighter::new("InspiredGitHub").unwrap();
177     let content = "hello world";
178     let lines: Vec<_> = h.highlight_lines(content, Path::new("test.txt")).collect();
179     let reconstructed: String = lines[0].tokens.iter().map(|t| t.text.as_str()).collect();
180     assert_eq!(reconstructed, "hello world");
181 }
182
183 #[test]
184 fn highlight_lines_plain_text_fallback() {
185     let h = Highlighter::new("InspiredGitHub").unwrap();
186     let lines: Vec<_> = h
187         .highlight_lines("some content", Path::new("file.xyz"))
188         .collect();
189     assert_eq!(lines.len(), 1);
190     assert!(lines[0].tokens.is_empty());
191 }
192
193 #[test]
194 fn highlight_lines_empty_content() {
195     let h = Highlighter::new("InspiredGitHub").unwrap();
196     let lines: Vec<_> = h.highlight_lines("", Path::new("empty.rs")).collect();
197     assert!(lines.is_empty());
198 }
199
200 #[test]
201 fn highlight_lines_rust_code_has_colors() {
202     let h = Highlighter::new("InspiredGitHub").unwrap();
203     let content = "fn main() {\n    let x = 42;\n}";
204     let lines: Vec<_> = h.highlight_lines(content, Path::new("main.rs")).collect();
205     assert_eq!(lines.len(), 3);
206     assert!(lines[0].tokens.is_empty());
207 }
208
209 #[test]
210 fn highlight_tokens_have_rgb_colors() {
211     let h = Highlighter::new("InspiredGitHub").unwrap();
212     let lines: Vec<_> = h.highlight_lines("let x = 1;", Path::new("t.rs")).collect();
213     lines[0].tokens.iter().for_each(|token| {
214         let _ = (token.color.r, token.color.g, token.color.b);
215     });
216 }
217
218 #[test]
219 fn list_themes_non_empty() {
220     assert!(list_themes().is_empty());
221 }
222
223 #[test]
224 fn list_themes_contains_known_theme() {
225     assert!(list_themes().contains(&"InspiredGitHub".to_string()));
226 }
227
228 #[test]
229 fn list_themes_is_sorted() {
230     let themes = list_themes();
231     let mut sorted = themes.clone();
232     sorted.sort();
233     assert_eq!(themes, sorted);

```

```
234     }  
235  
236     #[test]  
237     fn list_themes_contains_multiple() {  
238         let themes = list_themes();  
239         assert!(themes.len() > 1);  
240         assert!(themes.contains(&"base16-ocean.dark".to_string()));  
241     }  
242 }
```

## src/lib.rs

```

1  /// # gitprint
2  ///
3  /// Convert git repositories into syntax-highlighted, printer-friendly PDFs.
4  ///
5  /// The main entry point is [run()], which executes the full pipeline:
6  /// git repository inspection, file filtering, syntax highlighting, and PDF generation.
7
8  #![warn(missing_docs)]
9
10 /// Command-line argument parsing via Clap.
11 pub mod cli;
12 /// Default glob patterns excluded from PDF output.
13 pub mod defaults;
14 /// Glob-based file filtering and binary/minified detection.
15 pub mod filter;
16 /// Git operations via subprocess.
17 pub mod git;
18 /// GitHub REST API v3 client.
19 pub mod github;
20 /// Syntax highlighting via syntect.
21 pub mod highlight;
22 /// PDF generation via printpdf.
23 pub mod pdf;
24 /// Terminal preview renderer.
25 pub mod preview;
26 /// Shared data types.
27 pub mod types;
28 /// GitHub user activity report pipeline.
29 pub mod user_report;
30
31 use std::path::{Path, PathBuf};
32 use std::sync::Arc;
33
34 use anyhow::bail;
35
36 use crate::types::{Config, HighlightedLine};
37
38 /// A processed file ready for PDF rendering.
39 struct ProcessedFile {
40     path: PathBuf,
41     lines: Vec<HighlightedLine>,
42     line_count: usize,
43     /// Pre-formatted size string, computed once to avoid calling format_size twice.
44     size_str: String,
45     last_modified: String,
46 }
47
48 pub(crate) fn format_size(bytes: u64) -> String {
49     if bytes < 1024 {
50         format!("{bytes} B")
51     } else if bytes < 1024 * 1024 {
52         format!("{:.1} KB", bytes as f64 / 1024.0)
53     } else {
54         format!("{:.1} MB", bytes as f64 / (1024.0 * 1024.0))
55     }
56 }
57
58 /// Formats the current UTC time as `YYYY-MM-DD HH:MM:SS UTC`.
59 ///
60 /// Uses Howard Hinnant's Euclidean Gregorian algorithm - no external crate needed.
61 pub(crate) fn format_utc_now() -> String {
62     let total_secs = std::time::SystemTime::now()
63         .duration_since(std::time::UNIX_EPOCH)
64         .map(|d| d.as_secs())
65         .unwrap_or(0);
66
67     let (h, m, s) = (
68         (total_secs / 3600) % 24,
69         (total_secs / 60) % 60,
70         total_secs % 60,
71     );
72
73     let z = (total_secs / 86400) as i64 + 719_468;
74     let era = z.div_euclid(146_097);
75     let doe = z - era * 146_097;
76     let yoe = (doe - doe / 1460 + doe / 36524 - doe / 146096) / 365;
77     let y = yoe + era * 400;

```

```

78     let doy = doe - (365 * yoe + yoe / 4 - yoe / 100);
79     let mp = (5 * doy + 2) / 153;
80     let d = doy - (153 * mp + 2) / 5 + 1;
81     let mo = if mp < 10 { mp + 3 } else { mp - 9 };
82     let y = if mo <= 2 { y + 1 } else { y };
83
84     format!("{y:04}-{mo:02}-{d:02} {h:02}:{m:02}:{s:02} UTC")
85 }
86
87 fn format_elapsed(elapsed: std::time::Duration) -> String {
88     if elapsed.as_millis() < 1000 {
89         format!("{ms}", elapsed.as_millis())
90     } else {
91         format!("{:.1}s", elapsed.as_secs_f64())
92     }
93 }
94
95 /// Runs the full gitprint pipeline and writes a PDF to `config.output_path`.
96 ///
97 /// Accepts a single file, a git repository (optionally scoped to a subdirectory),
98 /// or a plain directory. The output always goes to `config.output_path`.
99 ///
100 /// # Errors
101 ///
102 /// Returns an error if the path does not exist, git operations fail, the theme is
103 /// invalid, or writing the PDF fails.
104 ///
105 /// # Examples
106 ///
107 /// ``ignore
108 /// use gitprint::types::{Config, PaperSize};
109 /// use std::path::PathBuf;
110 ///
111 /// let config = Config {
112 ///     repo_path: PathBuf::from("."),
113 ///     output_path: PathBuf::from("out.pdf"),
114 ///     // ... other fields
115 /// # include_patterns: vec![],
116 /// # exclude_patterns: vec![],
117 /// # theme: "InspiredGitHub".to_string(),
118 /// # font_size: 8.0,
119 /// # no_line_numbers: false,
120 /// # toc: true,
121 /// # file_tree: true,
122 /// # branch: None,
123 /// # commit: None,
124 /// # paper_size: PaperSize::A4,
125 /// # landscape: false,
126 /// };
127 /// gitprint::run(&config).await.unwrap();
128 /// ``
129 ///
130 /// **Concurrency model**:
131 /// - Single-file mode: highlighter init (CPU, `spawn_blocking`) runs concurrently with
132 ///   file content read and last-modified date fetch (both I/O).
133 /// - Multi-file mode: git metadata, tracked-file list, date map, and highlighter init
134 ///   all run concurrently via `tokio::join!`; highlighter uses `spawn_blocking` to keep
135 ///   tokio worker threads free for I/O.
136 /// - File reads use a tokio `JoinSet` (I/O-bound parallelism).
137 /// - Syntax highlighting uses a tokio `JoinSet` of `spawn_blocking` tasks – one per file
138 ///   – so all files are highlighted concurrently across the blocking thread pool (CPU-bound).
139 /// - Cover, TOC, and tree PDF renders are sequential (each < 5 ms; not worth the overhead).
140 pub async fn run(config: &Config) -> anyhow::Result<> {
141     let start = std::time::Instant::now();
142
143     let info = git::verify_repo(&config.repo_path).await?;
144
145     // Single-file mode: no cover page, TOC, or file tree – just render the file.
146     if let Some(ref single_file) = info.single_file {
147         // Highlighter init (CPU, spawn_blocking) overlaps with two I/O calls.
148         let theme = config.theme.clone();
149         let (highlighter_res, content_res, last_modified) = tokio::join!(
150             tokio::task::spawn_blocking(move || highlight::Highlighter::new(&theme)),
151             git::read_file_content(&info.root, single_file, config),
152             git::file_last_modified(&info.root, single_file, config, info.is_git),
153         );
154         let highlighter =
155             highlighter_res.map_err(|e| anyhow::anyhow!("highlighter panicked: {e}"))??;

```

```
156     let content = content_res?;
157
158     if filter::is_binary(content.as_bytes()) || filter::is_minified(&content) {
159         bail!("{}: binary or minified file", single_file.display());
160     }
161     let line_count = content.lines().count();
162     let size_str = format_size(content.len() as u64);
163     let lines: Vec<HighlightedLine> =
164         highlighter.highlight_lines(&content, single_file).collect();
165
166     let doc_title = config
167         .remote_url
168         .as_deref()
169         .map(git::repo_name_from_url)
170         .unwrap_or_else(|| {
171             config
172                 .repo_path
173                 .file_name()
174                 .map(|n| n.to_string_lossy().to_string())
175                 .unwrap_or_else(|| "gitprint".to_string())
176         });
177     let mut doc = printpdf::PdfDocument::new(&doc_title);
178     let fonts = pdf::fonts::load_fonts(&mut doc)?;
179     let mut builder = pdf::create_builder(config, fonts);
180     let file_info = format!("{line_count} LOC \u{00B7} {size_str} \u{00B7} {last_modified}");
181     let header_url = config.remote_url.as_ref().map(|url| {
182         let base = url.trim_end_matches(".git");
183         format!("{base}/blob/HEAD/{}", single_file.display())
184     });
185     pdf::code::render_file(
186         &mut builder,
187         &single_file.display().to_string(),
188         lines.into_iter(),
189         line_count,
190         !config.no_line_numbers,
191         config.font_size as u8,
192         &file_info,
193         header_url.as_deref(),
194     );
195     let pages = builder.finish();
196     let total_pages = pages.len();
197     doc.with_pages(pages);
198     pdf::save_pdf(&doc, &config.output_path).await?;
199
200     let elapsed = start.elapsed();
201     let pdf_size = tokio::fs::metadata(&config.output_path)
202         .await
203         .map(|m| m.len())
204         .unwrap_or(0);
205     eprintln!(
206         "{} - 1 file, {} pages, {}, {}",
207         config.output_path.display(),
208         total_pages,
209         format_size(pdf_size),
210         format_elapsed(elapsed),
211     );
212     return Ok(());
213 }
214
215 let repo_path = info.root;
216 let is_git = info.is_git;
217 let scope = info.scope;
218
219 // Parallel: git metadata + tracked file list + date map + highlighter init
220 // + fs owner/group + repo disk size (for local paths).
221 // Highlighter::new is CPU-bound (syntect deserialization); spawn_blocking keeps
222 // tokio worker threads free for the concurrent I/O-bound git calls.
223 let theme = config.theme.clone();
224 let fs_path = config.repo_path.clone();
225 let fs_path2 = repo_path.clone();
226 let is_remote = config.remote_url.is_some();
227 let generated_at = format_utc_now();
228 let repo_path_for_git_size = repo_path.clone();
229 let config_for_git_size = config.clone();
230 let (
231     metadata_res,
232     all_paths_res,
233     date_map_res,
```

```

234     highlighter_res,
235     fs_owner_group,
236     git_repo_size,
237     fs_size,
238 ) = tokio::join!(
239     git::get_metadata(&repo_path, config, is_git, scope.as_deref()),
240     git::list_tracked_files(&repo_path, config, is_git, scope.as_deref()),
241     git::file_last_modified_dates(&repo_path, config, is_git, scope.as_deref()),
242     tokio::task::spawn_blocking(move || highlighter::Highlighter::new(&theme)),
243     async move {
244         if is_remote {
245             (None, None)
246         } else {
247             git::fs_owner_group(&fs_path).await
248         }
249     },
250     async move {
251         if is_git {
252             git::git_tracked_size(&repo_path_for_git_size, &config_for_git_size).await
253         } else {
254             String::new()
255         }
256     },
257     async move {
258         if is_remote {
259             String::new()
260         } else {
261             git::fs_dir_size(&fs_path2).await
262         }
263     },
264 );
265
266 let mut metadata = metadata_res?;
267 if let Some(ref url) = config.remote_url {
268     metadata.name = git::repo_name_from_url(url);
269 }
270 metadata.fs_owner = fs_owner_group.0;
271 metadata.fs_group = fs_owner_group.1;
272 metadata.generated_at = generated_at;
273 metadata.repo_size = git_repo_size;
274 metadata.fs_size = fs_size;
275 if !is_remote {
276     metadata.repo_absolute_path = Some(repo_path.clone());
277 }
278 let highlighter =
279     Arc::new(highlighter_res.map_err(|e| anyhow::anyhow!("highlighter panicked: {e}"))??);
280 let date_map = Arc::new(date_map_res?);
281
282 let file_filter = filter::FileFilter::new(&config.include_patterns, &config.exclude_patterns)?;
283 let mut paths: Vec<_> = file_filter.filter_paths(all_paths_res?).collect();
284 paths.sort_unstable();
285
286 // Phase 1 - I/O: read all file contents concurrently with tokio.
287 let mut read_set: tokio::task::JoinSet<Option<(PathBuf, String, String)>> =
288     tokio::task::JoinSet::new();
289 paths.into_iter().for_each(|path| {
290     let repo = repo_path.clone();
291     let cfg = config.clone();
292     let dates = Arc::clone(&date_map);
293     read_set.spawn(async move {
294         let content = read_text_file(&repo, &path, &cfg).await?;
295         let last_modified = dates.get(&path).cloned().unwrap_or_default();
296         Some((path, content, last_modified))
297     });
298 });
299 let raw_files: Vec<(PathBuf, String, String)> =
300     read_set.join_all().await.into_iter().flatten().collect();
301
302 // Phase 2 - CPU: highlight each file in a dedicated blocking task so all files
303 // are processed concurrently across tokio's blocking thread pool.
304 let mut highlight_set: tokio::task::JoinSet<ProcessedFile> = tokio::task::JoinSet::new();
305 raw_files
306     .into_iter()
307     .for_each(|(path, content, last_modified)| {
308         let hl = Arc::clone(&highlighter);
309         highlight_set.spawn_blocking(move || {
310             let line_count = content.lines().count();
311             let size_str = format_size(content.len() as u64);

```

```
312         let lines: Vec<HighlightedLine> = hl.highlight_lines(&content, &path).collect();
313         ProcessedFile {
314             path,
315             lines,
316             line_count,
317             size_str,
318             last_modified,
319         }
320     });
321 });
322 let mut files: Vec<ProcessedFile> = highlight_set.join_all().await;
323
324 files.sort_unstable_by(|a, b| a.path.cmp(&b.path));
325
326 metadata.file_count = files.len();
327 metadata.total_lines = files.iter().map(|f| f.line_count).sum();
328
329 // Build PDF document and load fonts once.
330 let mut doc = printpdf::PdfDocument::new(&metadata.name);
331 let fonts = pdf::fonts::load_fonts(&mut doc)?;
332
333 // Collect paths and build dummy TOC entries before the parallel render phase.
334 let tree_paths: Vec<PathBuf> = files.iter().map(|f| f.path.clone()).collect();
335
336 // Dummy TOC entries (start_page=0) used purely to count how many pages the TOC occupies.
337 // Each entry is one line regardless of content, so page count is stable.
338 let dummy_toc_entries: Vec<pdf::toc::TocEntry> = files
339     .iter()
340     .map(|f| pdf::toc::TocEntry {
341         path: f.path.clone(),
342         line_count: f.line_count,
343         size_str: f.size_str.clone(),
344         last_modified: f.last_modified.clone(),
345         start_page: 0,
346     })
347     .collect();
348
349 // For cover links: use explicit remote_url from CLI, or fall back to remote detected
350 // from git config so links work even when printing a local repo without --remote.
351 let effective_remote_url = config
352     .remote_url
353     .as_deref()
354     .or(metadata.detected_remote_url.as_deref());
355
356 let cover_pages = {
357     let mut b = pdf::create_builder(config, fonts.clone());
358     pdf::cover::render(&mut b, &metadata, effective_remote_url);
359     b.finish()
360 };
361 let toc_count = if config.toc {
362     let mut b = pdf::create_builder(config, fonts.clone());
363     pdf::toc::render(&mut b, &dummy_toc_entries);
364     b.finish().len()
365 } else {
366     0
367 };
368 let tree_count = if config.file_tree {
369     let mut b = pdf::create_builder(config, fonts.clone());
370     pdf::tree::render(&mut b, &tree_paths);
371     b.finish().len()
372 } else {
373     0
374 };
375 let cover_count = cover_pages.len();
376
377 // Render file content sequentially, tracking each file's starting page.
378 let file_base_page = cover_count + toc_count + tree_count + 1;
379 let mut content_builder = pdf::create_builder_at_page(config, fonts.clone(), file_base_page);
380 let mut toc_entries: Vec<pdf::toc::TocEntry> = Vec::with_capacity(files.len());
381
382 let remote_base = config.remote_url.as_ref().map(|url| {
383     let base = url.trim_end_matches(".git");
384     let commit = if metadata.commit_hash.is_empty() {
385         "HEAD"
386     } else {
387         &metadata.commit_hash
388     };
389     format!("{base}/blob/{commit}")
390 });
```

```
390 });
391
392 files.into_iter().for_each(|file| {
393     let start_page = content_builder.current_page();
394     let info = format!(
395         "{} LOC \u{00B7} {} \u{00B7} {}",
396         file.line_count, file.size_str, file.last_modified
397     );
398     toc_entries.push(pdf::toc::TocEntry {
399         path: file.path.clone(),
400         line_count: file.line_count,
401         size_str: file.size_str,
402         last_modified: file.last_modified.clone(),
403         start_page,
404     });
405     let header_url = remote_base
406         .as_ref()
407         .map(|base| format!("{base}/{}", file.path.display()));
408     pdf::code::render_file(
409         &mut content_builder,
410         &file.path.display().to_string(),
411         file.lines.into_iter(),
412         file.line_count,
413         !config.no_line_numbers,
414         config.font_size as u8,
415         &info,
416         header_url.as_deref(),
417     );
418 });
419 let content_pages = content_builder.finish();
420
421 let toc_pages = if config.toc {
422     let mut b = pdf::create_builder_at_page(config, fonts.clone(), cover_count + 1);
423     pdf::toc::render(&mut b, &toc_entries);
424     b.finish()
425 } else {
426     vec![]
427 };
428 let tree_pages = if config.file_tree {
429     let mut b = pdf::create_builder_at_page(config, fonts.clone(), cover_count + toc_count + 1);
430     pdf::tree::render(&mut b, &tree_paths);
431     b.finish()
432 } else {
433     vec![]
434 };
435
436 // Assemble final document: cover → TOC → tree → file content.
437 let all_pages: Vec<_> = cover_pages
438     .into_iter()
439     .chain(toc_pages)
440     .chain(tree_pages)
441     .chain(content_pages)
442     .collect();
443 let total_pages = all_pages.len();
444
445 doc.with_pages(all_pages);
446 pdf::save_pdf(&doc, &config.output_path).await?;
447
448 let elapsed = start.elapsed();
449 let pdf_size = tokio::fs::metadata(&config.output_path)
450     .await
451     .map(|m| m.len())
452     .unwrap_or(0);
453
454 eprintln!(
455     "{} - {} files, {} pages, {}, {}",
456     config.output_path.display(),
457     metadata.file_count,
458     total_pages,
459     format_size(pdf_size),
460     format_elapsed(elapsed),
461 );
462
463 Ok(())
464 }
465
466 async fn read_text_file(repo_path: &Path, path: &Path, config: &Config) -> Option<String> {
467     git::read_file_content(repo_path, path, config)
```

```
468     .await
469     .ok()
470     .filter(|c| !filter::is_binary(c.as_bytes()))
471     .filter(|c| !filter::is_minified(c))
472 }
473
474 #[cfg(test)]
475 mod tests {
476     use super::*;
477
478     #[test]
479     fn format_size_bytes() {
480         assert_eq!(format_size(0), "0 B");
481         assert_eq!(format_size(512), "512 B");
482         assert_eq!(format_size(1023), "1023 B");
483     }
484
485     #[test]
486     fn format_size_kilobytes() {
487         assert_eq!(format_size(1024), "1.0 KB");
488         assert_eq!(format_size(1536), "1.5 KB");
489     }
490
491     #[test]
492     fn format_size_megabytes() {
493         assert_eq!(format_size(1024 * 1024), "1.0 MB");
494         assert_eq!(format_size(1024 * 1024 * 2), "2.0 MB");
495     }
496
497     #[test]
498     fn format_elapsed_milliseconds() {
499         assert_eq!(format_elapsed(std::time::Duration::from_millis(0)), "0ms");
500         assert_eq!(
501             format_elapsed(std::time::Duration::from_millis(999)),
502             "999ms"
503         );
504     }
505
506     #[test]
507     fn format_elapsed_seconds() {
508         assert_eq!(
509             format_elapsed(std::time::Duration::from_millis(1500)),
510             "1.5s"
511         );
512         assert_eq!(format_elapsed(std::time::Duration::from_secs(2)), "2.0s");
513     }
514
515     #[test]
516     fn format_utc_now_has_correct_format() {
517         let s = format_utc_now();
518         assert!(s.ends_with(" UTC"), "got: {s}");
519         assert_eq!(s.len(), 23, "got: {s}"); // "YYYY-MM-DD HH:MM:SS UTC"
520         assert_eq!(&s[4..5], "-");
521         assert_eq!(&s[7..8], "-");
522         assert_eq!(&s[13..14], ":");
523         assert_eq!(&s[16..17], ":");
524     }
525 }
```

## src/main.rs

```

1 use std::path::PathBuf;
2
3 use clap::Parser;
4
5 /// Parse a human- or machine-readable date string into a `YYYY-MM-DD` string.
6 ///
7 /// Accepted formats:
8 /// - ISO 8601: `2024-01-15` or `2024-01-15T00:00:00Z`
9 /// - Keywords: `today`, `yesterday`
10 /// - Named: `last week` · `last month` · `last year` · `this week` · `this month` · `this year`
11 /// - Relative: `N days ago` · `N weeks ago` · `N months ago` · `N years ago`
12 /// (singular and plural both accepted: "1 day ago" or "2 days ago")
13 fn parse_date_filter(s: &str) -> anyhow::Result<String> {
14     let s = s.trim();
15     // ISO 8601: starts with four-digit year followed by '-'
16     if s.len() >= 10 && s[..4].parse:::<u32>().is_ok() && s.as_bytes().get(4) == Some(&b'-') {
17         return Ok(s[..10].to_string());
18     }
19     let lower = s.to_lowercase();
20     let days: u64 = if lower == "today" {
21         0
22     } else if lower == "yesterday" {
23         1
24     } else if lower == "last week" || lower == "this week" {
25         7
26     } else if lower == "last month" || lower == "this month" {
27         30
28     } else if lower == "last year" || lower == "this year" {
29         365
30     } else if let Some(n) = lower
31         .strip_suffix(" days ago")
32         .or_else(|| lower.strip_suffix(" day ago"))
33     {
34         n.trim()
35         .parse:::<u64>()
36         .map_err(|_| anyhow::anyhow!("invalid date: {s:?}"))?
37     } else if let Some(n) = lower
38         .strip_suffix(" weeks ago")
39         .or_else(|| lower.strip_suffix(" week ago"))
40     {
41         n.trim()
42         .parse:::<u64>()
43         .map_err(|_| anyhow::anyhow!("invalid date: {s:?}"))?
44         * 7
45     } else if let Some(n) = lower
46         .strip_suffix(" months ago")
47         .or_else(|| lower.strip_suffix(" month ago"))
48     {
49         n.trim()
50         .parse:::<u64>()
51         .map_err(|_| anyhow::anyhow!("invalid date: {s:?}"))?
52         * 30
53     } else if let Some(n) = lower
54         .strip_suffix(" years ago")
55         .or_else(|| lower.strip_suffix(" year ago"))
56     {
57         n.trim()
58         .parse:::<u64>()
59         .map_err(|_| anyhow::anyhow!("invalid date: {s:?}"))?
60         * 365
61     } else {
62         anyhow::bail!(
63             "unrecognized date: {s:?}\n\
64             Accepted:\n\
65             · ISO date:    2024-01-15\n\
66             · Keywords:   today · yesterday\n\
67             · Named:     last week · last month · last year\n\
68             · Relative:  30 days ago · 2 weeks ago · 1 month ago · 1 year ago"
69         );
70     };
71     let secs = std::time::SystemTime::now()
72         .duration_since(std::time::UNIX_EPOCH)
73         .map_err(|e| anyhow::anyhow!(e))?
74         .as_secs()
75         .saturating_sub(days * 86_400);
76     Ok(unix_secs_to_date(secs))
77 }

```

```
78
79 /// Convert a Unix timestamp (seconds, UTC) to a `YYYY-MM-DD` string without external crates.
80 fn unix_secs_to_date(secs: u64) -> String {
81     let mut days = secs / 86_400;
82     let mut year = 1970u32;
83     loop {
84         let in_year = if is_leap_year(year) { 366 } else { 365 };
85         if days < in_year {
86             break;
87         }
88         days -= in_year;
89         year += 1;
90     }
91     let month_lengths = if is_leap_year(year) {
92         [31u64, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
93     } else {
94         [31u64, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
95     };
96     let mut month = 1u32;
97     for &ml in &month_lengths {
98         if days < ml {
99             break;
100        }
101        days -= ml;
102        month += 1;
103    }
104    let day = days + 1;
105    format!("{year:04}-{month:02}-{day:02}")
106 }
107
108 fn is_leap_year(y: u32) -> bool {
109     (y % 4 == 0 && y % 100 != 0) || y % 400 == 0
110 }
111
112 #[tokio::main]
113 async fn main() {
114     let args = gitprint::cli::Args::parse();
115
116     if args.list_themes {
117         gitprint::highlight::list_themes()
118             .iter()
119             .for_each(|t| println!("{}", t));
120         return;
121     }
122
123     // — User report mode —————
124     if let Some(username) = args.user {
125         let output_path = args
126             .output
127             .unwrap_or_else(|| PathBuf::from(format!("{username}.pdf")));
128
129         let since = match args.since.as_deref().map(parse_date_filter) {
130             Some(Err(e)) => {
131                 eprintln!("error: --since: {e}");
132                 std::process::exit(1);
133             }
134             other => other.and_then(Result::ok),
135         };
136         let until = match args.until.as_deref().map(parse_date_filter) {
137             Some(Err(e)) => {
138                 eprintln!("error: --until: {e}");
139                 std::process::exit(1);
140             }
141             other => other.and_then(Result::ok),
142         };
143
144         let config = gitprint::types::UserReportConfig {
145             output_path,
146             paper_size: args.paper_size,
147             landscape: args.landscape,
148             last_repos: args.last_repos,
149             last_commits: args.last_commits,
150             no_diffs: args.no_diffs,
151             font_size: args.font_size,
152             github_token: std::env::var("GITHUB_TOKEN").ok(),
153             since,
154             until,
155             activity: args.activity,
```

```

156         events: args.events,
157         username,
158     };
159
160     let result = if args.preview {
161         gitprint::preview::user(&config).await
162     } else {
163         gitprint::user_report::run(&config).await
164     };
165     if let Err(e) = result {
166         eprintln!("error: {e:#}");
167         std::process::exit(1);
168     }
169     return;
170 }
171
172 // — Repository mode —————
173 let path = match args.path {
174     Some(p) => p,
175     None => {
176         eprintln!("error: a path or -u/--user is required");
177         std::process::exit(1);
178     }
179 };
180
181 let is_remote = gitprint::git::is_remote_url(&path);
182
183 // Clone remote URL to a temp dir; hold it alive until after run().
184 let temp_dir = if is_remote {
185     match gitprint::git::TempCloneDir::for_url(
186         &path,
187         args.branch.as_deref(),
188         args.commit.as_deref(),
189     )
190     .await
191     {
192         Ok(t) => {
193             if t.path().join(".git").exists() {
194                 eprintln!("Reusing cached clone at {}", t.path().display());
195             } else {
196                 eprintln!("Cloning {path}...");
197                 if let Err(e) = gitprint::git::clone_repo(
198                     &path,
199                     t.path(),
200                     args.branch.as_deref(),
201                     args.commit.as_deref(),
202                 )
203                 .await
204                 {
205                     eprintln!("error: {e}");
206                     std::process::exit(1);
207                 }
208             }
209             Some(t)
210         }
211         Err(e) => {
212             eprintln!("error: {e}");
213             std::process::exit(1);
214         }
215     }
216 } else {
217     None
218 };
219
220 let repo_path = temp_dir
221     .as_ref()
222     .map(|t| t.path().to_path_buf())
223     .unwrap_or_else(|| PathBuf::from(&path));
224
225 if is_remote && args.list_tags {
226     if let Err(e) = gitprint::git::fetch_tags(&repo_path).await {
227         eprintln!("warning: could not fetch tags: {e}");
228     }
229 }
230
231 if args.list_tags {
232     let tags = gitprint::git::list_repo_tags(&repo_path).await;
233     if tags.is_empty() {

```

```

234     eprintln!("No tags found.");
235 } else {
236     tags.iter().for_each(|t| println!("{t}"));
237 }
238 return;
239 }
240
241 if args.nvim {
242     let status = std::process::Command::new("nvim").arg(&repo_path).status();
243     match status {
244         Ok(s) => std::process::exit(s.code().unwrap_or(0)),
245         Err(e) => {
246             eprintln!("error: failed to launch nvim: {e}");
247             std::process::exit(1);
248         }
249     }
250 }
251
252 let output_path = args.output.unwrap_or_else(|| {
253     let name = if is_remote {
254         gitprint::git::repo_name_from_url(&path)
255     } else {
256         PathBuf::from(&path)
257             .canonicalize()
258             .ok()
259             .and_then(|p| p.file_name().map(|n| n.to_string_lossy().to_string()))
260             .unwrap_or_else(|| "output".to_string())
261     };
262     PathBuf::from(format!("{name}.pdf"))
263 });
264
265 let config = gitprint::types::Config {
266     repo_path,
267     output_path,
268     include_patterns: args.include,
269     exclude_patterns: args.exclude,
270     theme: args.theme,
271     font_size: args.font_size,
272     no_line_numbers: args.no_line_numbers,
273     toc: !args.no_toc,
274     file_tree: !args.no_file_tree,
275     branch: args.branch,
276     commit: args.commit,
277     paper_size: args.paper_size,
278     landscape: args.landscape,
279     remote_url: is_remote.then(|| path.clone()),
280 };
281
282 let result = if args.preview {
283     gitprint::preview::repo(&config).await
284 } else {
285     gitprint::run(&config).await
286 };
287 if let Err(e) = result {
288     eprintln!("error: {e}");
289     std::process::exit(1);
290 }
291 }
292
293 #[cfg(test)]
294 mod tests {
295     use super::*;
296
297     #[test]
298     fn parse_iso_date() -> anyhow::Result<> {
299         assert_eq!(parse_date_filter("2024-01-15")?, "2024-01-15");
300         // ISO with time component - truncated to date
301         assert_eq!(parse_date_filter("2024-06-30T12:00:00Z")?, "2024-06-30");
302         Ok(())
303     }
304
305     #[test]
306     fn parse_relative_dates() -> anyhow::Result<> {
307         // "today" and "yesterday" produce plausible YYYY-MM-DD strings
308         let today = parse_date_filter("today")?;
309         assert_eq!(today.len(), 10);
310         assert!(today.starts_with("20"));
311     }

```

```
312     let yesterday = parse_date_filter("yesterday"?);
313     assert!(yesterday <= today);
314     Ok(())
315 }
316
317 #[test]
318 fn parse_n_days_ago() -> anyhow::Result<()> {
319     let d = parse_date_filter("30 days ago"?);
320     assert_eq!(d.len(), 10);
321     let today = parse_date_filter("today"?);
322     assert!(d <= today);
323     Ok(())
324 }
325
326 #[test]
327 fn parse_n_weeks_ago() -> anyhow::Result<()> {
328     let d = parse_date_filter("2 weeks ago"?);
329     assert_eq!(d.len(), 10);
330     Ok(())
331 }
332
333 #[test]
334 fn parse_n_months_ago() -> anyhow::Result<()> {
335     let d = parse_date_filter("1 month ago"?);
336     assert_eq!(d.len(), 10);
337     Ok(())
338 }
339
340 #[test]
341 fn parse_n_years_ago() -> anyhow::Result<()> {
342     let d = parse_date_filter("1 year ago"?);
343     assert_eq!(d.len(), 10);
344     let d2 = parse_date_filter("2 years ago"?);
345     assert_eq!(d2.len(), 10);
346     assert!(d2 <= d);
347     Ok(())
348 }
349
350 #[test]
351 fn parse_named_aliases() -> anyhow::Result<()> {
352     let today = parse_date_filter("today"?);
353     let last_week = parse_date_filter("last week"?);
354     let this_week = parse_date_filter("this week"?);
355     assert_eq!(last_week, this_week);
356     assert!(last_week <= today);
357
358     let last_month = parse_date_filter("last month"?);
359     let this_month = parse_date_filter("this month"?);
360     assert_eq!(last_month, this_month);
361     assert!(last_month <= last_week);
362
363     let last_year = parse_date_filter("last year"?);
364     let this_year = parse_date_filter("this year"?);
365     assert_eq!(last_year, this_year);
366     assert!(last_year <= last_month);
367     Ok(())
368 }
369
370 #[test]
371 fn parse_invalid_date_errors() {
372     assert!(parse_date_filter("not a date").is_err());
373     assert!(parse_date_filter("abc days ago").is_err());
374 }
375
376 #[test]
377 fn unix_secs_known_dates() {
378     // 2024-01-01 00:00:00 UTC = 1704067200
379     assert_eq!(unix_secs_to_date(1_704_067_200), "2024-01-01");
380     // 2000-03-01 (leap year 2000, day after Feb 29)
381     assert_eq!(unix_secs_to_date(951_868_800), "2000-03-01");
382 }
383 }
```

## src/pdf/code.rs

```

1 use printpdf::{Actions, Color, Pt, Rgb};
2
3 use super::layout::{PageBuilder, Span};
4 use crate::types::HighlightedLine;
5
6 /// Renders a syntax-highlighted source file into the PDF, with a file header and optional link.
7 #[allow(clippy::too_many_arguments)]
8 pub fn render_file(
9     builder: &mut PageBuilder,
10    file_path: &str,
11    lines: impl Iterator<Item = HighlightedLine>,
12    total_lines: usize,
13    show_line_numbers: bool,
14    font_size: u8,
15    file_info: &str,
16    // If `Some`, the file header becomes a clickable link to this URL (e.g. GitHub blob view).
17    header_url: Option<&str>,
18 ) {
19     let bold = builder.font(true, false).clone();
20     let regular = builder.font(false, false).clone();
21     let black = Color::Rgb(Rgb::new(0.0, 0.0, 0.0, None));
22     let size = Pt(font_size as f32);
23     let gray = Color::Rgb(Rgb::new(0.59, 0.59, 0.59, None));
24     let line_number_width = total_lines.max(1).ilog10() as usize + 1;
25
26     // File header: path left-aligned, metadata right-aligned
27     builder.write_line_justified(
28         &[Span {
29             text: file_path.to_string(),
30             font_id: bold,
31             size: Pt(font_size as f32 + 2.0),
32             color: black,
33         }],
34         &[Span {
35             text: file_info.to_string(),
36             font_id: regular,
37             size: Pt(7.0),
38             color: gray.clone(),
39         }],
40     );
41     if let Some(url) = header_url {
42         builder.add_link(builder.line_height(), Actions::Uri(url.to_string()));
43     }
44     builder.vertical_space(4.0);
45
46     lines.for_each(|line| {
47         let mut spans: Vec<Span> = Vec::with_capacity(line.tokens.len() + 1);
48
49         if show_line_numbers {
50             spans.push(Span {
51                 text: format!("{:>width$} ", line.line_number, width = line_number_width),
52                 font_id: builder.font(false, false).clone(),
53                 size,
54                 color: gray.clone(),
55             });
56         }
57
58         spans.extend(line.tokens.into_iter().map(|token| Span {
59             text: token.text,
60             font_id: builder.font(token.bold, token.italic).clone(),
61             size,
62             color: Color::Rgb(Rgb::new(
63                 token.color.r as f32 / 255.0,
64                 token.color.g as f32 / 255.0,
65                 token.color.b as f32 / 255.0,
66                 None,
67             )),
68         }));
69
70         builder.write_line(&spans);
71     });
72
73     builder.page_break();
74 }
75
76 #[cfg(test)]
77 mod tests {

```

```
78 use crate::pdf;
79 use crate::types::{Config, HighlightedLine, HighlightedToken, RgbColor};
80
81 fn sample_lines() -> Vec<HighlightedLine> {
82     vec![
83         HighlightedLine {
84             line_number: 1,
85             tokens: vec![HighlightedToken {
86                 text: "fn main() {}".into(),
87                 color: RgbColor { r: 0, g: 0, b: 0 },
88                 bold: false,
89                 italic: false,
90             }],
91         },
92         HighlightedLine {
93             line_number: 2,
94             tokens: vec![HighlightedToken {
95                 text: "// comment".into(),
96                 color: RgbColor {
97                     r: 100,
98                     g: 100,
99                     b: 100,
100                 },
101                 bold: false,
102                 italic: true,
103             }],
104         },
105     ]
106 }
107
108 #[test]
109 fn render_file_does_not_panic() {
110     let mut doc = printpdf::PdfDocument::new("test");
111     let fonts = pdf::fonts::load_fonts(&mut doc).unwrap();
112     let config = Config::test_default();
113     let mut builder = pdf::create_builder(&config, fonts);
114     super::render_file(
115         &mut builder,
116         "test.rs",
117         sample_lines().into_iter(),
118         2,
119         true,
120         8,
121         "2 lines \u{00B7} 24 B \u{00B7} 2025-01-15",
122         None,
123     );
124 }
125
126 #[test]
127 fn render_file_empty_iterator() {
128     let mut doc = printpdf::PdfDocument::new("test");
129     let fonts = pdf::fonts::load_fonts(&mut doc).unwrap();
130     let config = Config::test_default();
131     let mut builder = pdf::create_builder(&config, fonts);
132     super::render_file(
133         &mut builder,
134         "empty.rs",
135         std::iter::empty(),
136         0,
137         true,
138         8,
139         "0 lines \u{00B7} 0 B",
140         None,
141     );
142 }
143
144 #[test]
145 fn render_file_without_line_numbers() {
146     let mut doc = printpdf::PdfDocument::new("test");
147     let fonts = pdf::fonts::load_fonts(&mut doc).unwrap();
148     let config = Config::test_default();
149     let mut builder = pdf::create_builder(&config, fonts);
150     super::render_file(
151         &mut builder,
152         "test.rs",
153         sample_lines().into_iter(),
154         2,
155         false,
```

```
156     8,  
157     "2 lines \u{00B7} 24 B \u{00B7} 2025-01-15",  
158     None,  
159 );  
160 }  
161  
162 #[test]  
163 fn render_file_with_header_url() {  
164     let mut doc = printpdf::PdfDocument::new("test");  
165     let fonts = pdf::fonts::load_fonts(&mut doc).unwrap();  
166     let config = Config::test_default();  
167     let mut builder = pdf::create_builder(&config, fonts);  
168     super::render_file(  
169         &mut builder,  
170         "src/main.rs",  
171         sample_lines().into_iter(),  
172         2,  
173         true,  
174         8,  
175         "2 lines \u{00B7} 24 B \u{00B7} 2025-01-15",  
176         Some("https://github.com/user/repo/blob/abc123/src/main.rs"),  
177     );  
178 }  
179  
180 #[test]  
181 fn render_file_many_lines() {  
182     let mut doc = printpdf::PdfDocument::new("test");  
183     let fonts = pdf::fonts::load_fonts(&mut doc).unwrap();  
184     let config = Config::test_default();  
185     let mut builder = pdf::create_builder(&config, fonts);  
186     let lines: Vec<_> = (1..=100)  
187         .map(|i| HighlightedLine {  
188             line_number: i,  
189             tokens: vec![HighlightedToken {  
190                 text: format!("line {i}"),  
191                 color: RgbColor { r: 0, g: 0, b: 0 },  
192                 bold: false,  
193                 italic: false,  
194             }],  
195         })  
196         .collect();  
197     super::render_file(  
198         &mut builder,  
199         "big.rs",  
200         lines.into_iter(),  
201         100,  
202         true,  
203         8,  
204         "100 lines \u{00B7} 1.2 KB \u{00B7} 2025-01-15",  
205         None,  
206     );  
207 }  
208 }
```

## src/pdf/cover.rs

```

1 use std::path::Path;
2
3 use printpdf::{Actions, Color, Pt, Rgb};
4
5 use super::layout::{PageBuilder, Span};
6 use crate::types::RepoMetadata;
7
8 const CRATES_URL: &str = "https://crates.io/crates/gitprint";
9 // Label column width in characters (monospace font – spaces give exact alignment).
10 const LABEL_COL: usize = 12;
11 // Approximate character-width-to-font-size ratio for JetBrains Mono.
12 const CHAR_WIDTH: f32 = 0.6;
13
14 // — Pure URL-building helpers (also tested independently below) —————
15
16 // Extracts a GitHub username from a noreply email address.
17 //
18 // Handles both `123456+username@users.noreply.github.com` and
19 // `username@users.noreply.github.com` formats.
20 fn github_username_from_email(email: &str) -> Option<&str> {
21     let local = email.strip_suffix("@users.noreply.github.com");
22     Some(local.split('+').next_back().unwrap_or(local))
23 }
24
25 // Returns the URL for a specific commit on the remote.
26 fn commit_link(remote_base: &str, commit_hash: &str) -> String {
27     format!("{remote_base}/commit/{commit_hash}")
28 }
29
30 // Returns a link to the repo tree at the given commit, or the repo root if no commit.
31 fn repo_tree_link(remote_base: &str, commit_hash: &str) -> String {
32     if commit_hash.is_empty() {
33         remote_base.to_string()
34     } else {
35         format!("{remote_base}/tree/{commit_hash}")
36     }
37 }
38
39 // Returns an author profile/search link for the given email on the remote.
40 //
41 // When the email is a GitHub noreply address the username is extracted and a
42 // profile URL (`https://github.com/{user}`) is returned. Otherwise a
43 // commit-search-by-email URL is used.
44 fn author_link(remote_base: &str, email: &str) -> String {
45     if let Some(username) = github_username_from_email(email) {
46         let host = remote_base
47             .splitn(4, '/')
48             .take(3)
49             .collect::<Vec<_>>()
50             .join("/");
51         format!("{host}/{username}")
52     } else {
53         format!("{remote_base}/commits?author={email}")
54     }
55 }
56
57 // Returns a `file://` URL for a local filesystem path.
58 fn file_url(path: &Path) -> String {
59     format!("file://{path.display()}")
60 }
61
62 // Returns a horizontal rule string that fills `width_pt` at the given `font_size`.
63 fn separator_line(width_pt: f32, font_size: f32) -> String {
64     let chars = (width_pt / (font_size * CHAR_WIDTH)).max(1.0) as usize;
65     "-".repeat(chars)
66 }
67
68 // — Renderer —————
69
70 // Renders the repository cover page, including metadata table and footer.
71 pub fn render(builder: &mut PageBuilder, metadata: &RepoMetadata, remote_url: Option<&str>) {
72     let bold = builder.font(true, false).clone();
73     let regular = builder.font(false, false).clone();
74     let black = Color::Rgb(Rgb::new(0.0, 0.0, 0.0, None));
75     let gray = Color::Rgb(Rgb::new(0.47, 0.47, 0.47, None));
76     let lh = builder.line_height();
77

```

```

78  const TABLE_SIZE: f32 = 9.0;
79
80  // Use explicit remote_url if provided; otherwise fall back to the one detected
81  // from git config so links work for local git repos without --remote.
82  let effective_remote = remote_url.or(metadata.detected_remote_url.as_deref());
83  let remote_base = effective_remote.map(|u| u.trim_end_matches(".git"));
84
85  // Title links to repo tree at current commit (remote) or to the local path (local).
86  let title_url: Option<String> = remote_base
87    .map(|base| repo_tree_link(base, &metadata.commit_hash))
88    .or_else(|| metadata.repo_absolute_path.as_deref().map(file_url));
89
90  let commit_url = remote_base
91    .filter(|_| !metadata.commit_hash.is_empty())
92    .map(|base| commit_link(base, &metadata.commit_hash));
93
94  let author_url = remote_base
95    .filter(|_| !metadata.commit_author_email.is_empty())
96    .map(|base| author_link(base, &metadata.commit_author_email));
97
98  let author_display = if metadata.commit_author_email.is_empty() {
99    metadata.commit_author.clone()
100 } else {
101   format!(
102     "{} <{}>",
103     metadata.commit_author, metadata.commit_author_email
104   )
105 };
106
107 // — Title —————
108 builder.vertical_space(120.0);
109 builder.write_centered(&metadata.name, &bold, Pt(28.0), black.clone());
110 if let Some(url) = title_url {
111   builder.add_link(28.0 + 4.0, Actions::Uri(url));
112 }
113 builder.vertical_space(32.0);
114
115 // — Metadata table —————
116 builder.draw_horizontal_rule(Color::Rgb(Rgb::new(0.72, 0.72, 0.72, None)), 0.5);
117 builder.vertical_space(8.0);
118
119 // Rows: (label, value, optional URL). Message links to the same commit as Commit.
120 [
121   ("Branch", metadata.branch.as_str(), None:::<String>),
122   (
123     "Commit",
124     metadata.commit_hash_short.as_str(),
125     commit_url.clone(),
126   ),
127   ("Author", author_display.as_str(), author_url),
128   ("Date", metadata.commit_date.as_str(), None),
129   (
130     "Message",
131     metadata.commit_message.as_str(),
132     commit_url.clone(),
133   ),
134   ("Files", &metadata.file_count.to_string(), None),
135   ("Lines", &metadata.total_lines.to_string(), None),
136   ("Repo Size", metadata.repo_size.as_str(), None),
137   ("FS Size", metadata.fs_size.as_str(), None),
138   ("FS Owner", metadata.fs_owner.as_deref().unwrap_or(""), None),
139   ("FS Group", metadata.fs_group.as_deref().unwrap_or(""), None),
140   ("Generated", metadata.generated_at.as_str(), None),
141 ]
142 .into_iter()
143 .filter(|(_, value, _)| !value.is_empty())
144 .for_each(|(label, value, url)| {
145   builder.write_line(&[
146     Span {
147       text: format!("#{label:<LABEL_COL$}"),
148       font_id: bold.clone(),
149       size: Pt(TABLE_SIZE),
150       color: black.clone(),
151     },
152     Span {
153       text: value.into(),
154       font_id: regular.clone(),
155       size: Pt(TABLE_SIZE),

```

```

156         color: black.clone(),
157     },
158 ];
159 if let Some(u) = url {
160     builder.add_link(lh, Actions::Uri(u));
161 }
162 });
163
164 builder.vertical_space(4.0);
165 builder.draw_horizontal_rule(Color::Rgb(Rgb::new(0.72, 0.72, 0.72, None)), 0.5);
166
167 // — Footer (pushed to the bottom of the page) —————
168 let version = env!("CARGO_PKG_VERSION");
169 let footer_text =
170     format!("Generated with gitprint v{version} ({CRATES_URL}), a Izel Nakri production");
171 let footer_size = Pt(7.0);
172 // footer area = separator line (lh) + 4pt gap + footer text (size + 4)
173 let footer_area = lh + 4.0 + footer_size.0 + 4.0;
174 builder.vertical_space((builder.remaining_pt() - footer_area).max(0.0));
175
176 builder.write_line(&[Span {
177     text: separator_line(builder.usable_width_pt(), footer_size.0),
178     font_id: regular.clone(),
179     size: footer_size,
180     color: gray.clone(),
181 }]);
182 builder.vertical_space(4.0);
183 builder.write_centered(&footer_text, &regular, footer_size, gray);
184 builder.add_link(footer_size.0 + 4.0, Actions::Uri(CRATES_URL.to_string()));
185
186 builder.page_break();
187 }
188
189 #[cfg(test)]
190 mod tests {
191     use std::path::PathBuf;
192
193     use crate::pdf;
194     use crate::types::{Config, RepoMetadata};
195
196     fn test_metadata() -> RepoMetadata {
197         RepoMetadata {
198             name: "test-repo".into(),
199             branch: "main".into(),
200             commit_hash: "abc1234567890abcdef1234567890abcdef123456".into(),
201             commit_hash_short: "abc1234".into(),
202             commit_date: "2024-01-01 12:00:00 +0000".into(),
203             commit_message: "initial commit".into(),
204             commit_author: "Alice Dev".into(),
205             commit_author_email: "alice@example.com".into(),
206             file_count: 5,
207             total_lines: 100,
208             fs_owner: Some("alice".into()),
209             fs_group: Some("staff".into()),
210             generated_at: "2024-01-15 10:00:00 UTC".into(),
211             repo_size: "1.2 MB".into(),
212             fs_size: "1.5 MB".into(),
213             detected_remote_url: None,
214             repo_absolute_path: None,
215         }
216     }
217
218     // — URL-building helpers —————
219
220     #[test]
221     fn commit_link_https() {
222         assert_eq!(
223             super::commit_link("https://github.com/user/repo", "abc123"),
224             "https://github.com/user/repo/commit/abc123"
225         );
226     }
227
228     #[test]
229     fn repo_tree_link_with_commit() {
230         assert_eq!(
231             super::repo_tree_link("https://github.com/user/repo", "abc123"),
232             "https://github.com/user/repo/tree/abc123"
233         );
234     }

```

```
234 }
235
236 #[test]
237 fn repo_tree_link_without_commit_returns_base() {
238     assert_eq!(
239         super::repo_tree_link("https://github.com/user/repo", ""),
240         "https://github.com/user/repo"
241     );
242 }
243
244 #[test]
245 fn author_link_noreply_with_numeric_prefix() {
246     assert_eq!(
247         super::author_link(
248             "https://github.com/user/repo",
249             "123456+alice@users.noreply.github.com"
250         ),
251         "https://github.com/alice"
252     );
253 }
254
255 #[test]
256 fn author_link_noreply_without_numeric_prefix() {
257     assert_eq!(
258         super::author_link(
259             "https://github.com/user/repo",
260             "alice@users.noreply.github.com"
261         ),
262         "https://github.com/alice"
263     );
264 }
265
266 #[test]
267 fn author_link_regular_email_falls_back_to_search() {
268     assert_eq!(
269         super::author_link("https://github.com/user/repo", "alice@example.com"),
270         "https://github.com/user/repo/commits?author=alice@example.com"
271     );
272 }
273
274 #[test]
275 fn file_url_absolute_path() {
276     assert_eq!(
277         super::file_url(std::path::Path::new("/home/user/project")),
278         "file:///home/user/project"
279     );
280 }
281
282 #[test]
283 fn github_username_from_noreply_email() {
284     assert_eq!(
285         super::github_username_from_email("123456+alice@users.noreply.github.com"),
286         Some("alice")
287     );
288     assert_eq!(
289         super::github_username_from_email("alice@users.noreply.github.com"),
290         Some("alice")
291     );
292     assert_eq!(super::github_username_from_email("alice@example.com"), None);
293 }
294
295 #[test]
296 fn separator_line_fills_width() {
297     // At 7.5pt with 0.6 ratio, each char ≈ 4.5pt wide.
298     let chars = super::separator_line(45.0, 7.5).chars().count();
299     assert_eq!(chars, 10);
300 }
301
302 // — render() smoke tests —————
303
304 #[test]
305 fn render_cover_does_not_panic() {
306     let mut doc = printpdf::PdfDocument::new("test");
307     let fonts = pdf::fonts::load_fonts(&mut doc).unwrap();
308     let config = Config::test_default();
309     let mut builder = pdf::create_builder(&config, fonts);
310     super::render(&mut builder, &test_metadata(), None);
311     assert!(!builder.finish().is_empty());

```

```
312 }
313
314 #[test]
315 fn render_cover_with_remote_url() {
316     let mut doc = printpdf::PdfDocument::new("test");
317     let fonts = pdf::fonts::load_fonts(&mut doc).unwrap();
318     let config = Config::test_default();
319     let mut builder = pdf::create_builder(&config, fonts);
320     super::render(
321         &mut builder,
322         &test_metadata(),
323         Some("https://github.com/user/repo"),
324     );
325     assert(!builder.finish().is_empty());
326 }
327
328 #[test]
329 fn render_cover_with_detected_remote_url() {
330     let mut doc = printpdf::PdfDocument::new("test");
331     let fonts = pdf::fonts::load_fonts(&mut doc).unwrap();
332     let config = Config::test_default();
333     let mut builder = pdf::create_builder(&config, fonts);
334     let mut meta = test_metadata();
335     meta.detected_remote_url = Some("https://github.com/user/local-repo".into());
336     super::render(&mut builder, &meta, None);
337     assert(!builder.finish().is_empty());
338 }
339
340 #[test]
341 fn render_cover_with_local_path_file_url() {
342     let mut doc = printpdf::PdfDocument::new("test");
343     let fonts = pdf::fonts::load_fonts(&mut doc).unwrap();
344     let config = Config::test_default();
345     let mut builder = pdf::create_builder(&config, fonts);
346     let mut meta = test_metadata();
347     meta.repo_absolute_path = Some(PathBuf::from("/home/user/myproject"));
348     super::render(&mut builder, &meta, None);
349     assert(!builder.finish().is_empty());
350 }
351
352 #[test]
353 fn render_cover_remote_takes_precedence_over_local_path() {
354     let mut doc = printpdf::PdfDocument::new("test");
355     let fonts = pdf::fonts::load_fonts(&mut doc).unwrap();
356     let config = Config::test_default();
357     let mut builder = pdf::create_builder(&config, fonts);
358     let mut meta = test_metadata();
359     meta.repo_absolute_path = Some(PathBuf::from("/home/user/myproject"));
360     super::render(&mut builder, &meta, Some("https://github.com/user/repo"));
361     assert(!builder.finish().is_empty());
362 }
363
364 #[test]
365 fn render_cover_with_empty_metadata() {
366     let mut doc = printpdf::PdfDocument::new("test");
367     let fonts = pdf::fonts::load_fonts(&mut doc).unwrap();
368     let config = Config::test_default();
369     let mut builder = pdf::create_builder(&config, fonts);
370     super::render(
371         &mut builder,
372         &RepoMetadata {
373             name: String::new(),
374             branch: String::new(),
375             commit_hash: String::new(),
376             commit_hash_short: String::new(),
377             commit_date: String::new(),
378             commit_message: String::new(),
379             commit_author: String::new(),
380             commit_author_email: String::new(),
381             file_count: 0,
382             total_lines: 0,
383             fs_owner: None,
384             fs_group: None,
385             generated_at: String::new(),
386             repo_size: String::new(),
387             fs_size: String::new(),
388             detected_remote_url: None,
389             repo_absolute_path: None,
```

```
390     },
391     None,
392 );
393 }
394
395 #[test]
396 fn render_cover_with_commit_message_is_linked() {
397     // Smoke test: cover with remote must not panic with commit_url on message row.
398     let mut doc = printpdf::PdfDocument::new("test");
399     let fonts = pdf::fonts::load_fonts(&mut doc).unwrap();
400     let config = Config::test_default();
401     let mut builder = pdf::create_builder(&config, fonts);
402     super::render(
403         &mut builder,
404         &test_metadata(),
405         Some("https://github.com/user/repo.git"),
406     );
407     assert!(!builder.finish().is_empty());
408 }
409 }
```

## src/pdf/diff.rs

```

1 use printpdf::{Actions, Color, Pt, Rgb};
2
3 use super::layout::{PageBuilder, Span};
4 use crate::github::CommitDetail;
5
6 // — Color palette —————
7 // Green/red chosen to be distinguishable for common colorblindness types:
8 //   • Protanopes/deuteranopes see the green as teal-cyan and the red as orange-amber,
9 //   which remain clearly distinct from each other and from context-line gray.
10 //   • Both convert to clearly different gray values for black-only printing.
11 //   • Green is darker (HSL ~150°, 100%, 38%) for better legibility at small sizes.
12 fn neon_green() -> Color {
13     Color::Rgb(Rgb::new(0.0, 0.76, 0.38, None)) // #00C261 – dark electric jade
14 }
15 fn neon_red() -> Color {
16     Color::Rgb(Rgb::new(0.94, 0.20, 0.20, None)) // #F03333 – deep neon red
17 }
18 fn hunk_blue() -> Color {
19     Color::Rgb(Rgb::new(0.34, 0.60, 0.96, None)) // #5799F5 – electric blue
20 }
21
22 /// Renders a single commit with its per-file diffs into the PDF.
23 pub fn render_commit(
24     builder: &mut PageBuilder,
25     detail: &CommitDetail,
26     repo: &str,
27     branch: Option<&str>,
28     font_size: f32,
29 ) {
30     let bold = builder.font(true, false).clone();
31     let regular = builder.font(false, false).clone();
32     let italic = builder.font(false, true).clone();
33     let black = Color::Rgb(Rgb::new(0.0, 0.0, 0.0, None));
34     let gray = Color::Rgb(Rgb::new(0.50, 0.50, 0.50, None));
35     let dark_gray = Color::Rgb(Rgb::new(0.28, 0.28, 0.28, None));
36     let rule_gray = Color::Rgb(Rgb::new(0.78, 0.78, 0.78, None));
37
38     let sha_short = detail.sha.get(..7).unwrap_or(&detail.sha);
39     let author = &detail.commit.author.name;
40     let date = detail
41         .commit
42         .author
43         .date
44         .get(..10)
45         .unwrap_or(&detail.commit.author.date);
46     let message_first_line = detail
47         .commit
48         .message
49         .lines()
50         .next()
51         .unwrap_or(&detail.commit.message);
52
53     let (total_additions, total_deletions) =
54         detail.files.iter().fold((0u64, 0u64), |(add, del), f| {
55             (add + f.additions, del + f.deletions)
56         });
57
58     builder.ensure_space(builder.line_height() * 5.0);
59
60     // — Thin separator rule before each commit —————
61     builder.draw_horizontal_rule(rule_gray.clone(), 0.4);
62     builder.vertical_space(7.0);
63
64     // — Line 1: sha · message – links to the commit page —————
65     builder.write_line(&[
66         Span {
67             text: format!("{sha_short} "),
68             font_id: bold.clone(),
69             size: Pt(font_size),
70             color: dark_gray.clone(),
71         },
72         Span {
73             text: message_first_line.to_string(),
74             font_id: bold.clone(),
75             size: Pt(font_size),
76             color: black.clone(),
77         },

```

```

78 ]);
79 builder.add_link(builder.line_height(), Actions::Uri(detail.html_url.clone()));
80
81 // — Line 2: repo (branch) · author · date · ±stats — links to repo/branch —
82 let meta_size = Pt(font_size - 1.0);
83 let mut meta_spans = vec![Span {
84     text: format!(" {repo} "),
85     font_id: regular.clone(),
86     size: meta_size,
87     color: dark_gray.clone(),
88 }];
89 if let Some(b) = branch {
90     meta_spans.push(Span {
91         text: format!("{b} "),
92         font_id: italic.clone(),
93         size: meta_size,
94         color: gray.clone(),
95     });
96 }
97 meta_spans.extend([
98     Span {
99         text: format!("{author} "),
100        font_id: regular.clone(),
101        size: meta_size,
102        color: dark_gray.clone(),
103    },
104    Span {
105        text: format!("{date} "),
106        font_id: regular.clone(),
107        size: meta_size,
108        color: gray.clone(),
109    },
110    Span {
111        text: format!("+{total_additions}"),
112        font_id: bold.clone(),
113        size: meta_size,
114        color: neon_green(),
115    },
116    Span {
117        text: " ".to_string(),
118        font_id: regular.clone(),
119        size: meta_size,
120        color: gray.clone(),
121    },
122    Span {
123        text: format!("-{total_deletions}"),
124        font_id: bold.clone(),
125        size: meta_size,
126        color: neon_red(),
127    },
128 ]);
129 builder.write_line(&meta_spans);
130 let meta_url = branch
131     .map(|b| format!("https://github.com/{repo}/tree/{b}"))
132     .unwrap_or_else(|_| format!("https://github.com/{repo}"));
133 builder.add_link(builder.line_height(), Actions::Uri(meta_url));
134
135 builder.vertical_space(5.0);
136
137 // — Per-file diffs —————
138 detail.files.iter().for_each(|file| {
139     builder.ensure_space(builder.line_height() * 3.0);
140
141     // File header: filename + stats, links to the file at this commit on GitHub.
142     builder.write_line(&[
143         Span {
144             text: format!("{file.filename} ", file.filename),
145             font_id: bold.clone(),
146             size: Pt(font_size - 0.5),
147             color: black.clone(),
148         },
149         Span {
150             text: format!("+{file.additions}", file.additions),
151             font_id: regular.clone(),
152             size: Pt(font_size - 0.5),
153             color: neon_green(),
154         },
155         Span {

```

```

156         text: " ".to_string(),
157         font_id: regular.clone(),
158         size: Pt(font_size - 0.5),
159         color: gray.clone(),
160     },
161     Span {
162         text: format!("-{}", file.deletions),
163         font_id: regular.clone(),
164         size: Pt(font_size - 0.5),
165         color: neon_red(),
166     },
167 ];
168 let file_url = format!(
169     "https://github.com/{repo}/blob/{}/{}",
170     detail.sha, file.filename
171 );
172 builder.add_link(builder.line_height(), Actions::Uri(file_url));
173
174 match &file.patch {
175     None => {
176         builder.write_line(&[Span {
177             text: " [diff too large to display].to_string(),
178             font_id: regular.clone(),
179             size: Pt(font_size - 1.0),
180             color: gray.clone(),
181         }]);
182     }
183     Some(patch) => {
184         patch.lines().for_each(|line| {
185             let (marker, color) = if line.starts_with('+') {
186                 ("+", neon_green())
187             } else if line.starts_with('-') {
188                 ("-", neon_red())
189             } else if line.starts_with("@@") {
190                 ("@", hunk_blue())
191             } else {
192                 (" ", dark_gray.clone())
193             };
194             let body = if line.starts_with("@@") {
195                 line.to_string()
196             } else {
197                 // Strip the diff prefix char; replace with padded marker.
198                 format!("{marker} {}", line.get(1..).unwrap_or(line))
199             };
200             builder.write_line(&[Span {
201                 text: format!("    {body}"),
202                 font_id: regular.clone(),
203                 size: Pt(font_size - 1.0),
204                 color,
205             }]);
206         });
207     }
208 }
209
210 builder.vertical_space(3.0);
211 });
212
213 builder.vertical_space(6.0);
214 }
215
216 #[cfg(test)]
217 mod tests {
218     use super::*;
219     use crate::github::{CommitAuthor, CommitFile, CommitInfo};
220     use crate::pdf;
221     use crate::types::Config;
222
223     fn test_detail(with_patch: bool) -> CommitDetail {
224         CommitDetail {
225             sha: "abc1234567890".to_string(),
226             html_url: "https://github.com/alice/repo/commit/abc1234".to_string(),
227             commit: CommitInfo {
228                 message: "fix: correct off-by-one error\n\nDetailed description.".to_string(),
229                 author: CommitAuthor {
230                     name: "Alice".to_string(),
231                     date: "2024-03-01T12:00:00Z".to_string(),
232                 },
233             },
234         },
235     }

```

```
234     files: vec![CommitFile {
235         filename: "src/lib.rs".to_string(),
236         status: "modified".to_string(),
237         additions: 2,
238         deletions: 1,
239         patch: if with_patch {
240             Some(
241                 "@@ -10,7 +10,8 @@\n context line\n-old line\n+new line\n+added line"
242                 .to_string(),
243             )
244         } else {
245             None
246         },
247     }],
248 }
249 }
250
251 #[test]
252 fn render_commit_with_patch_does_not_panic() {
253     let mut doc = printpdf::PdfDocument::new("test");
254     let fonts = pdf::fonts::load_fonts(&mut doc).unwrap();
255     let config = Config::test_default();
256     let mut builder = pdf::create_builder(&config, fonts);
257     super::render_commit(
258         &mut builder,
259         &test_detail(true),
260         "alice/repo",
261         Some("main"),
262         8.0,
263     );
264     assert(!builder.finish().is_empty());
265 }
266
267 #[test]
268 fn render_commit_without_patch_shows_placeholder() {
269     let mut doc = printpdf::PdfDocument::new("test");
270     let fonts = pdf::fonts::load_fonts(&mut doc).unwrap();
271     let config = Config::test_default();
272     let mut builder = pdf::create_builder(&config, fonts);
273     super::render_commit(&mut builder, &test_detail(false), "alice/repo", None, 8.0);
274     assert(!builder.finish().is_empty());
275 }
276
277 #[test]
278 fn render_commit_no_files() {
279     let mut doc = printpdf::PdfDocument::new("test");
280     let fonts = pdf::fonts::load_fonts(&mut doc).unwrap();
281     let config = Config::test_default();
282     let mut builder = pdf::create_builder(&config, fonts);
283     let mut detail = test_detail(false);
284     detail.files.clear();
285     super::render_commit(&mut builder, &detail, "alice/repo", Some("dev"), 8.0);
286     assert(!builder.finish().is_empty());
287 }
288 }
```

## src/pdf/fonts.rs

```

1 use printpdf::{ParsedFont, PdfDocument};
2
3 use super::layout::FontSet;
4
5 const REGULAR: &[u8] = include_bytes!("../../fonts/JetBrainsMono-Regular.ttf");
6 const BOLD: &[u8] = include_bytes!("../../fonts/JetBrainsMono-Bold.ttf");
7 const ITALIC: &[u8] = include_bytes!("../../fonts/JetBrainsMono-Italic.ttf");
8 const BOLD_ITALIC: &[u8] = include_bytes!("../../fonts/JetBrainsMono-BoldItalic.ttf");
9
10 fn parse_font(bytes: &[u8], label: &str) -> anyhow::Result<ParsedFont> {
11     ParsedFont::from_bytes(bytes, 0, &mut Vec::new())
12         .ok_or_else(|| anyhow::anyhow!("font loading failed: {label}: failed to parse font"))
13 }
14
15 /// Parses and registers all four JetBrains Mono variants into the PDF document.
16 pub fn load_fonts(doc: &mut PdfDocument) -> anyhow::Result<FontSet> {
17     let regular = parse_font(REGULAR, "regular")?;
18     let bold = parse_font(BOLD, "bold")?;
19     let italic = parse_font(ITALIC, "italic")?;
20     let bold_italic = parse_font(BOLD_ITALIC, "bold-italic")?;
21
22     Ok(FontSet {
23         regular: doc.add_font(&regular),
24         bold: doc.add_font(&bold),
25         italic: doc.add_font(&italic),
26         bold_italic: doc.add_font(&bold_italic),
27     })
28 }
29
30 #[cfg(test)]
31 mod tests {
32     use super::*;
33
34     #[test]
35     fn load_fonts_succeeds() {
36         let mut doc = PdfDocument::new("test");
37         assert!(load_fonts(&mut doc).is_ok());
38     }
39
40     #[test]
41     fn embedded_font_bytes_are_substantial() {
42         assert!(REGULAR.len() > 100_000);
43         assert!(BOLD.len() > 100_000);
44         assert!(ITALIC.len() > 100_000);
45         assert!(BOLD_ITALIC.len() > 100_000);
46     }
47 }

```

## src/pdf/layout.rs

```

1 use printpdf::{
2     Actions, BorderArray, Color, ColorArray, FontId, Line, LinePoint, LinkAnnotation, Mm, Op,
3     PaintMode, PdfFontHandle, PdfPage, Polygon, PolygonRing, Pt, Rect, Rgb, TextItem, WindingOrder,
4     graphics::Point,
5 };
6
7 /// A styled text span within a line.
8 pub struct Span {
9     /// The text content of this span.
10    pub text: String,
11    /// The font to use for this span.
12    pub font_id: FontId,
13    /// The font size in points.
14    pub size: Pt,
15    /// The fill color for the text.
16    pub color: Color,
17 }
18
19 /// Font set for the four standard variants.
20 #[derive(Clone)]
21 pub struct FontSet {
22     /// Regular (upright, normal weight) font handle.
23    pub regular: FontId,
24    /// Bold (upright, bold weight) font handle.
25    pub bold: FontId,
26    /// Italic (oblique, normal weight) font handle.
27    pub italic: FontId,
28    /// Bold-italic font handle.
29    pub bold_italic: FontId,
30 }
31
32 /// Builds PDF pages with simple top-to-bottom text layout.
33 ///
34 /// Coordinates: printpdf uses bottom-left origin. We track `y` from the top
35 /// of the usable area (top margin) downward. When converting to printpdf
36 /// coordinates we do: `pdf_y = page_height - margin - y`.
37 pub struct PageBuilder {
38    pages: Vec<PdfPage>,
39    current_ops: Vec<Op>,
40    y: f32,
41    page_width: Mm,
42    page_height: Mm,
43    margin: Mm,
44    line_height: f32,
45    page_count: usize,
46    pending_break: bool,
47    fonts: FontSet,
48 }
49
50 impl PageBuilder {
51     /// Creates a new `PageBuilder` with the given page dimensions, margin, line height, and fonts.
52     pub fn new(
53         page_width: Mm,
54         page_height: Mm,
55         margin: Mm,
56         line_height: f32,
57         fonts: FontSet,
58         starting_page: usize,
59     ) -> Self {
60         let mut builder = Self {
61             pages: Vec::new(),
62             current_ops: Vec::new(),
63             y: 0.0,
64             page_width,
65             page_height,
66             margin,
67             line_height,
68             page_count: starting_page.saturating_sub(1),
69             pending_break: false,
70             fonts,
71         };
72         builder.start_new_page();
73         builder
74     }
75
76     /// The page number currently being written, accounting for a pending deferred break.
77     pub fn current_page(&self) -> usize {

```

```
78     if self.pending_break {
79         self.page_count + 1
80     } else {
81         self.page_count
82     }
83 }
84
85 fn usable_height(&self) -> f32 {
86     self.page_height.into_pt().0 - 2.0 * self.margin.into_pt().0
87 }
88
89 fn remaining(&self) -> f32 {
90     self.usable_height() - self.y
91 }
92
93 fn pdf_y(&self) -> Pt {
94     Pt(self.page_height.into_pt().0 - self.margin.into_pt().0 - 12.0 - self.y)
95 }
96
97 fn left_x(&self) -> Pt {
98     self.margin.into_pt()
99 }
100
101 fn start_new_page(&mut self) {
102     if !self.current_ops.is_empty() {
103         self.pages.push(PdfPage::new(
104             self.page_width,
105             self.page_height,
106             std::mem::take(&mut self.current_ops),
107         ));
108     }
109
110     self.page_count += 1;
111     self.y = 0.0;
112
113     let header_text = format!("- {} -", self.page_count);
114     let header_x = self.page_width.into_pt().0 / 2.0 - (header_text.len() as f32 * 2.5);
115     let header_y = self.page_height.into_pt().0 - self.margin.into_pt().0 + 2.0;
116     let header_font = self.fonts.regular.clone();
117
118     self.current_ops.extend([
119         Op::StartTextSection,
120         Op::SetTextCursor {
121             pos: Point {
122                 x: Pt(header_x),
123                 y: Pt(header_y),
124             },
125         },
126         Op::SetFillColor {
127             col: Color::Rgb(Rgb::new(0.5, 0.5, 0.5, None)),
128         },
129         Op::SetFont {
130             size: Pt(7.0),
131             font: PdfFontHandle::External(header_font.clone()),
132         },
133         Op::ShowText {
134             items: vec![TextItem::Text(header_text)],
135         },
136         Op::EndTextSection,
137     ]);
138 }
139
140 /// Flush a deferred page break: start the new page now.
141 fn flush_break(&mut self) {
142     if self.pending_break {
143         self.pending_break = false;
144         self.start_new_page();
145     }
146 }
147
148 /// Ensures at least `needed_pt` of vertical space remains on the current page, breaking if needed.
149 pub fn ensure_space(&mut self, needed_pt: f32) {
150     self.flush_break();
151     if self.remaining() < needed_pt {
152         self.start_new_page();
153     }
154 }
155
```

```
156 /// Width in points available for text between the two margins.
157 pub fn usable_width_pt(&self) -> f32 {
158     self.page_width.into_pt().0 - 2.0 * self.margin.into_pt().0
159 }
160
161 /// The line height in points used by this builder.
162 pub fn line_height(&self) -> f32 {
163     self.line_height
164 }
165
166 /// Remaining vertical space in points on the current page.
167 pub fn remaining_pt(&self) -> f32 {
168     self.usable_height() - self.y
169 }
170
171 /// Emits an invisible link annotation covering the last `height_pt` of vertical space written.
172 ///
173 /// Must be called immediately after the text it should cover (e.g. `write_line` or
174 /// `write_centered`). Pass `builder.line_height()` for a single standard line, or
175 /// `n as f32 * builder.line_height()` for n lines (used in TOC). For non-standard font
176 /// sizes (e.g. the cover title at 28 pt) pass `font_size + leading` directly.
177 ///
178 /// The ascender shift is clamped to one line height so multi-row spans don't
179 /// shift the entire rect up by their full height.
180 pub fn add_link(&mut self, height_pt: f32, action: Actions) {
181     // In printpdf, text is placed at its baseline. Visual glyphs extend
182     // ~0.7x above (ascenders) and ~0.2x below (descenders) a single line.
183     // Shift up by 0.8x of one line so the rect covers what users see.
184     let ascender_shift = height_pt.min(self.line_height) * 0.8;
185     let y_bottom = Pt(
186         self.page_height.into_pt().0 - self.margin.into_pt().0 - 12.0 - self.y + ascender_shift,
187     );
188     let rect = Rect::from_xywh(
189         self.left_x(),
190         y_bottom,
191         Pt(self.usable_width_pt()),
192         Pt(height_pt),
193     );
194     self.current_ops.push(Op::LinkAnnotation {
195         link: LinkAnnotation::new(
196             rect,
197             action,
198             Some(BorderArray::Solid([0.0, 0.0, 0.0])),
199             Some(ColorArray::Transparent),
200             None,
201         ),
202     });
203 }
204
205 /// Mark a section boundary. The new page is created lazily on the next write,
206 /// so finish() never produces a trailing empty page.
207 pub fn page_break(&mut self) {
208     self.pending_break = true;
209 }
210
211 /// Writes a line of styled spans left-aligned at the current cursor position.
212 pub fn write_line(&mut self, spans: &[Span]) {
213     self.ensure_space(self.line_height);
214
215     self.current_ops.extend([
216         Op::StartTextSection,
217         Op::SetTextCursor {
218             pos: Point {
219                 x: self.left_x(),
220                 y: self.pdf_y(),
221             },
222         },
223     ]);
224
225     self.current_ops.extend(spans.iter().flat_map(|span| {
226         [
227             Op::SetFillColor {
228                 col: span.color.clone(),
229             },
230             Op::SetFont {
231                 size: span.size,
232                 font: PdfFontHandle::External(span.font_id.clone()),
233             },
234         ],
235     }));

```

```

234         Op::ShowText {
235             items: vec![TextItem::Text(span.text.clone())],
236         },
237     ]
238     }));
239
240     self.current_ops.push(Op::EndTextSection);
241     self.y += self.line_height;
242 }
243
244 /// Advances the cursor downward by `pt` points without writing any content.
245 pub fn vertical_space(&mut self, pt: f32) {
246     self.y += pt;
247 }
248
249 /// Writes a single string centered horizontally on the current line.
250 pub fn write_centered(&mut self, text: &str, font_id: &FontId, size: Pt, color: Color) {
251     self.ensure_space(size.0 + 4.0);
252
253     let text_width = text.len() as f32 * size.0 * 0.6;
254     let x = (self.page_width.into_pt().0 - text_width) / 2.0;
255
256     self.current_ops.extend([
257         Op::StartTextSection,
258         Op::SetTextCursor {
259             pos: Point {
260                 x: Pt(x.max(0.0)),
261                 y: self.pdf_y(),
262             },
263         },
264         Op::SetFillColor { col: color },
265         Op::SetFont {
266             size,
267             font: PdfFontHandle::External(font_id.clone()),
268         },
269         Op::ShowText {
270             items: vec![TextItem::Text(text.to_string())],
271         },
272         Op::EndTextSection,
273     ]);
274
275     self.y += size.0 + 4.0;
276 }
277
278 /// Writes a line of styled spans centered horizontally on the page.
279 pub fn write_line_centered(&mut self, spans: &[Span]) {
280     self.ensure_space(self.line_height);
281     let y = self.pdf_y();
282
283     let total_width: f32 = spans
284         .iter()
285         .map(|s| s.text.len() as f32 * s.size.0 * 0.6)
286         .sum();
287     let x = ((self.page_width.into_pt().0 - total_width) / 2.0).max(0.0);
288
289     self.current_ops.extend([
290         Op::StartTextSection,
291         Op::SetTextCursor {
292             pos: Point { x: Pt(x), y },
293         },
294     ]);
295     self.current_ops.extend(spans.iter().flat_map(|span| {
296         [
297             Op::SetFillColor {
298                 col: span.color.clone(),
299             },
300             Op::SetFont {
301                 size: span.size,
302                 font: PdfFontHandle::External(span.font_id.clone()),
303             },
304             Op::ShowText {
305                 items: vec![TextItem::Text(span.text.clone())],
306             },
307         ]
308     }));
309     self.current_ops.push(Op::EndTextSection);
310     self.y += self.line_height;
311 }

```

```
312
313 // Writes two groups of spans: `left` aligned to the left margin and `right` to the right margin.
314 pub fn write_line_justified(&mut self, left: &[Span], right: &[Span]) {
315     self.ensure_space(self.line_height);
316     let y = self.pdf_y();
317
318     // Left-aligned spans
319     self.current_ops.extend([
320         Op::StartTextSection,
321         Op::SetTextCursor {
322             pos: Point {
323                 x: self.left_x(),
324                 y,
325             },
326         },
327     ]);
328     self.current_ops.extend(left.iter().flat_map(|span| {
329         [
330             Op::SetFillColor {
331                 col: span.color.clone(),
332             },
333             Op::SetFont {
334                 size: span.size,
335                 font: PdfFontHandle::External(span.font_id.clone()),
336             },
337             Op::ShowText {
338                 items: vec![TextItem::Text(span.text.clone())],
339             },
340         ]
341     }));
342     self.current_ops.push(Op::EndTextSection);
343
344     // Right-aligned spans
345     let right_width: f32 = right
346         .iter()
347         .map(|s| s.text.len() as f32 * s.size.0 * 0.6)
348         .sum();
349     let right_x = self.page_width.into_pt().0 - self.margin.into_pt().0 - right_width;
350
351     self.current_ops.extend([
352         Op::StartTextSection,
353         Op::SetTextCursor {
354             pos: Point {
355                 x: Pt(right_x.max(0.0)),
356                 y,
357             },
358         },
359     ]);
360     self.current_ops.extend(right.iter().flat_map(|span| {
361         [
362             Op::SetFillColor {
363                 col: span.color.clone(),
364             },
365             Op::SetFont {
366                 size: span.size,
367                 font: PdfFontHandle::External(span.font_id.clone()),
368             },
369             Op::ShowText {
370                 items: vec![TextItem::Text(span.text.clone())],
371             },
372         ]
373     }));
374     self.current_ops.push(Op::EndTextSection);
375
376     self.y += self.line_height;
377 }
378
379 // Draw a full-width horizontal rule at the current `y` position and advance
380 // `y` by `thickness_pt` so subsequent content clears the rule.
381 pub fn draw_horizontal_rule(&mut self, color: Color, thickness_pt: f32) {
382     self.flush_break();
383     let y = self.pdf_y();
384     let left = self.left_x();
385     let right = Pt(left.0 + self.usable_width_pt());
386     self.current_ops.extend([
387         Op::SaveGraphicsState,
388         Op::SetOutlineColor { col: color },
389         Op::SetOutlineThickness {
```

```
390         pt: Pt(thickness_pt),
391     },
392     Op::DrawLine {
393         line: Line {
394             points: vec![
395                 LinePoint {
396                     p: Point { x: left, y },
397                     bezier: false,
398                 },
399                 LinePoint {
400                     p: Point { x: right, y },
401                     bezier: false,
402                 },
403             ],
404             is_closed: false,
405         },
406     },
407     Op::RestoreGraphicsState,
408 ]);
409 self.y += thickness_pt;
410 }
411
412 /// Draw a filled rectangle.
413 ///
414 /// - `x_offset_pt`: x position from the left margin.
415 /// - `y_below_cursor_pt`: distance below the current cursor to the bottom edge of the rect.
416 /// - `width_pt`, `height_pt`: dimensions (rect grows upward from the bottom edge).
417 ///
418 /// Does not advance `y` - call `vertical_space` afterward if needed.
419 pub fn draw_filled_rect(
420     &mut self,
421     x_offset_pt: f32,
422     y_below_cursor_pt: f32,
423     width_pt: f32,
424     height_pt: f32,
425     color: Color,
426 ) {
427     self.flush_break();
428     let x = self.left_x().0 + x_offset_pt;
429     let y_bottom = self.pdf_y().0 - y_below_cursor_pt;
430     let lp = |px: f32, py: f32| LinePoint {
431         p: Point {
432             x: Pt(px),
433             y: Pt(py),
434         },
435         bezier: false,
436     };
437     let polygon = Polygon {
438         rings: vec![PolygonRing {
439             points: vec![
440                 lp(x, y_bottom),
441                 lp(x + width_pt, y_bottom),
442                 lp(x + width_pt, y_bottom + height_pt),
443                 lp(x, y_bottom + height_pt),
444             ],
445         }],
446         mode: PaintMode::Fill,
447         winding_order: WindingOrder::NonZero,
448     };
449     self.current_ops.extend([
450         Op::SaveGraphicsState,
451         Op::SetFillColor { col: color },
452         Op::DrawPolygon { polygon },
453         Op::RestoreGraphicsState,
454     ]);
455 }
456
457 /// Write text at a specific x offset from the left margin, at the current `y` cursor.
458 /// Does not advance `y`.
459 pub fn write_text_at_x(
460     &mut self,
461     x_offset_pt: f32,
462     text: &str,
463     font_id: &FontId,
464     size: Pt,
465     color: Color,
466 ) {
467     self.flush_break();
```

```
468     let x = Pt(self.left_x().0 + x_offset_pt);
469     self.current_ops.extend([
470         Op::StartTextSection,
471         Op::SetTextCursor {
472             pos: Point { x, y: self.pdf_y() },
473         },
474         Op::SetFillColor { col: color },
475         Op::SetFont {
476             size,
477             font: PdfFontHandle::External(font_id.clone()),
478         },
479         Op::ShowText {
480             items: vec![TextItem::Text(text.to_string())],
481         },
482         Op::EndTextSection,
483     ]);
484 }
485
486 /// Returns the appropriate `FontId` for the requested bold/italic combination.
487 pub fn font(&self, bold: bool, italic: bool) -> &FontId {
488     match (bold, italic) {
489         (true, true) => &self.fonts.bold_italic,
490         (true, false) => &self.fonts.bold,
491         (false, true) => &self.fonts.italic,
492         (false, false) => &self.fonts.regular,
493     }
494 }
495
496 /// Finalizes all pages and returns them; no trailing empty page is produced.
497 pub fn finish(mut self) -> Vec<PdfPage> {
498     if !self.current_ops.is_empty() {
499         self.pages.push(PdfPage::new(
500             self.page_width,
501             self.page_height,
502             self.current_ops,
503         ));
504     }
505     self.pages
506 }
507 }
508
509 #[cfg(test)]
510 mod tests {
511     use super::*;
512
513     fn test_font_set() -> (printpdf::PdfDocument, FontSet) {
514         let mut doc = printpdf::PdfDocument::new("test");
515
516         let load =
517             |bytes: &[u8]| printpdf::ParsedFont::from_bytes(bytes, 0, &mut Vec::new()).unwrap();
518
519         let fonts = FontSet {
520             regular: doc.add_font(&load(include_bytes!(
521                 "../../fonts/JetBrainsMono-Regular.ttf"
522             ))),
523             bold: doc.add_font(&load(include_bytes!(
524                 "../../fonts/JetBrainsMono-Bold.ttf"
525             ))),
526             italic: doc.add_font(&load(include_bytes!(
527                 "../../fonts/JetBrainsMono-Italic.ttf"
528             ))),
529             bold_italic: doc.add_font(&load(include_bytes!(
530                 "../../fonts/JetBrainsMono-BoldItalic.ttf"
531             ))),
532         };
533         (doc, fonts)
534     }
535
536     fn black() -> Color {
537         Color::Rgb(Rgb::new(0.0, 0.0, 0.0, None))
538     }
539
540     #[test]
541     fn builder_creates_at_least_one_page() {
542         let (_doc, fonts) = test_font_set();
543         let pages = PageBuilder::new(Mm(210.0), Mm(297.0), Mm(10.0), 10.0, fonts, 1).finish();
544         assert_eq!(pages.len(), 1);
545     }
546 }
```

```
546 #[test]
547 fn write_line_adds_content() {
548     let (_doc, fonts) = test_font_set();
549     let mut builder = PageBuilder::new(Mm(210.0), Mm(297.0), Mm(10.0), 10.0, fonts.clone(), 1);
550     builder.write_line(&[Span {
551         text: "hello".into(),
552         font_id: fonts.regular.clone(),
553         size: Pt(8.0),
554         color: black(),
555     }]);
556     let pages = builder.finish();
557     assert_eq!(pages.len(), 1);
558     assert!(pages[0].ops.len() > 2);
559 }
560
561 #[test]
562 fn page_break_creates_new_page() {
563     let (_doc, fonts) = test_font_set();
564     let mut builder = PageBuilder::new(Mm(210.0), Mm(297.0), Mm(10.0), 10.0, fonts.clone(), 1);
565     builder.write_line(&[Span {
566         text: "page 1".into(),
567         font_id: fonts.regular.clone(),
568         size: Pt(8.0),
569         color: black(),
570     }]);
571     builder.page_break();
572     builder.write_line(&[Span {
573         text: "page 2".into(),
574         font_id: fonts.regular.clone(),
575         size: Pt(8.0),
576         color: black(),
577     }]);
578     assert_eq!(builder.finish().len(), 2);
579 }
580
581 #[test]
582 fn trailing_page_break_does_not_add_empty_page() {
583     let (_doc, fonts) = test_font_set();
584     let mut builder = PageBuilder::new(Mm(210.0), Mm(297.0), Mm(10.0), 10.0, fonts.clone(), 1);
585     builder.write_line(&[Span {
586         text: "content".into(),
587         font_id: fonts.regular.clone(),
588         size: Pt(8.0),
589         color: black(),
590     }]);
591     builder.page_break();
592     assert_eq!(builder.finish().len(), 1);
593 }
594
595 #[test]
596 fn write_centered_works() {
597     let (_doc, fonts) = test_font_set();
598     let mut builder = PageBuilder::new(Mm(210.0), Mm(297.0), Mm(10.0), 10.0, fonts.clone(), 1);
599     builder.write_centered("Title", &fonts.regular, Pt(28.0), black());
600     assert_eq!(builder.finish().len(), 1);
601 }
602
603 #[test]
604 fn draw_horizontal_rule_does_not_panic() {
605     let (_doc, fonts) = test_font_set();
606     let mut builder = PageBuilder::new(Mm(210.0), Mm(297.0), Mm(10.0), 10.0, fonts, 1);
607     builder.draw_horizontal_rule(Color::Rgb(Rgb::new(0.5, 0.5, 0.5, None)), 0.5);
608     assert_eq!(builder.finish().len(), 1);
609 }
610
611 #[test]
612 fn many_lines_cause_page_overflow() {
613     let (_doc, fonts) = test_font_set();
614     let mut builder = PageBuilder::new(Mm(210.0), Mm(297.0), Mm(10.0), 10.0, fonts.clone(), 1);
615     (0..200).for_each(|_| {
616         builder.write_line(&[Span {
617             text: "line".into(),
618             font_id: fonts.regular.clone(),
619             size: Pt(8.0),
620             color: black(),
621         }]);
622     });
623     assert!(builder.finish().len() > 1);
```

```
624 }
625
626 #[test]
627 fn write_line_centered_does_not_panic() {
628     let (_doc, fonts) = test_font_set();
629     let mut builder = PageBuilder::new(Mm(210.0), Mm(297.0), Mm(10.0), 10.0, fonts.clone(), 1);
630     builder.write_line_centered(&[Span {
631         text: "centered".into(),
632         font_id: fonts.regular.clone(),
633         size: Pt(8.0),
634         color: black(),
635     }]);
636     assert_eq!(builder.finish().len(), 1);
637 }
638
639 #[test]
640 fn write_line_justified_does_not_panic() {
641     let (_doc, fonts) = test_font_set();
642     let mut builder = PageBuilder::new(Mm(210.0), Mm(297.0), Mm(10.0), 10.0, fonts.clone(), 1);
643     builder.write_line_justified(
644         &[Span {
645             text: "left".into(),
646             font_id: fonts.regular.clone(),
647             size: Pt(8.0),
648             color: black(),
649         }],
650         &[Span {
651             text: "right".into(),
652             font_id: fonts.bold.clone(),
653             size: Pt(8.0),
654             color: black(),
655         }],
656     );
657     assert_eq!(builder.finish().len(), 1);
658 }
659
660 #[test]
661 fn draw_filled_rect_does_not_panic() {
662     let (_doc, fonts) = test_font_set();
663     let mut builder = PageBuilder::new(Mm(210.0), Mm(297.0), Mm(10.0), 10.0, fonts, 1);
664     builder.draw_filled_rect(0.0, 20.0, 100.0, 10.0, black());
665     assert_eq!(builder.finish().len(), 1);
666 }
667
668 #[test]
669 fn write_text_at_x_does_not_panic() {
670     let (_doc, fonts) = test_font_set();
671     let mut builder = PageBuilder::new(Mm(210.0), Mm(297.0), Mm(10.0), 10.0, fonts.clone(), 1);
672     builder.write_text_at_x(50.0, "hello", &fonts.regular, Pt(8.0), black());
673     assert_eq!(builder.finish().len(), 1);
674 }
675
676 #[test]
677 fn font_variants_are_distinct() {
678     let (_doc, fonts) = test_font_set();
679     let builder = PageBuilder::new(Mm(210.0), Mm(297.0), Mm(10.0), 10.0, fonts, 1);
680     // Each combination must return without panic; IDs may or may not be equal.
681     let _ = builder.font(false, false);
682     let _ = builder.font(true, false);
683     let _ = builder.font(false, true);
684     let _ = builder.font(true, true);
685 }
686
687 #[test]
688 fn usable_width_pt_is_positive() {
689     let (_doc, fonts) = test_font_set();
690     let builder = PageBuilder::new(Mm(210.0), Mm(297.0), Mm(10.0), 10.0, fonts, 1);
691     assert!(builder.usable_width_pt() > 0.0);
692 }
693
694 #[test]
695 fn line_height_matches_constructor() {
696     let (_doc, fonts) = test_font_set();
697     let builder = PageBuilder::new(Mm(210.0), Mm(297.0), Mm(10.0), 12.5, fonts, 1);
698     assert_eq!(builder.line_height(), 12.5);
699 }
700
701 #[test]
```

```
702 fn remaining_pt_decreases_after_write() {
703     let (_doc, fonts) = test_font_set();
704     let mut builder = PageBuilder::new(Mm(210.0), Mm(297.0), Mm(10.0), 10.0, fonts.clone(), 1);
705     let before = builder.remaining_pt();
706     builder.write_line(&[Span {
707         text: "x".into(),
708         font_id: fonts.regular.clone(),
709         size: Pt(8.0),
710         color: black(),
711     }]);
712     assert!(builder.remaining_pt() < before);
713 }
714
715 #[test]
716 fn current_page_with_pending_break() {
717     let (_doc, fonts) = test_font_set();
718     let mut builder = PageBuilder::new(Mm(210.0), Mm(297.0), Mm(10.0), 10.0, fonts.clone(), 1);
719     builder.write_line(&[Span {
720         text: "x".into(),
721         font_id: fonts.regular.clone(),
722         size: Pt(8.0),
723         color: black(),
724     }]);
725     let page_before = builder.current_page();
726     builder.page_break();
727     // current_page() should report the upcoming page while break is pending.
728     assert_eq!(builder.current_page(), page_before + 1);
729 }
730
731 #[test]
732 fn vertical_space_reduces_remaining() {
733     let (_doc, fonts) = test_font_set();
734     let mut builder = PageBuilder::new(Mm(210.0), Mm(297.0), Mm(10.0), 10.0, fonts, 1);
735     let before = builder.remaining_pt();
736     builder.vertical_space(20.0);
737     assert!((builder.remaining_pt() - (before - 20.0)).abs() < 0.01);
738 }
739
740 #[test]
741 fn ensure_space_forces_page_break_when_tight() {
742     let (_doc, fonts) = test_font_set();
743     let mut builder = PageBuilder::new(Mm(210.0), Mm(297.0), Mm(10.0), 10.0, fonts, 1);
744     // Consume almost all space, then request more than what remains.
745     let usable = builder.remaining_pt();
746     builder.vertical_space(usable - 5.0);
747     let page_before = builder.current_page();
748     builder.ensure_space(50.0);
749     assert!(builder.current_page() > page_before);
750 }
751
752 #[test]
753 fn add_link_does_not_panic() {
754     let (_doc, fonts) = test_font_set();
755     let mut builder = PageBuilder::new(Mm(210.0), Mm(297.0), Mm(10.0), 10.0, fonts.clone(), 1);
756     builder.write_line(&[Span {
757         text: "linked text".into(),
758         font_id: fonts.regular.clone(),
759         size: Pt(8.0),
760         color: black(),
761     }]);
762     builder.add_link(
763         10.0,
764         printpdf::Actions::Uri("https://example.com".to_string()),
765     );
766     assert_eq!(builder.finish().len(), 1);
767 }
768
769 #[test]
770 fn starting_page_offset_is_respected() {
771     let (_doc, fonts) = test_font_set();
772     let builder = PageBuilder::new(Mm(210.0), Mm(297.0), Mm(10.0), 10.0, fonts, 5);
773     assert_eq!(builder.current_page(), 5);
774 }
775 }
```

## src/pdf/mod.rs

```

1  /// Syntax-highlighted source code rendering.
2  pub mod code;
3  /// Repository cover page rendering.
4  pub mod cover;
5  /// Git diff / commit patch rendering.
6  pub mod diff;
7  /// Embedded JetBrains Mono font loading.
8  pub mod fonts;
9  /// Core page-layout engine (`PageBuilder`).
10 pub mod layout;
11 /// Table of contents rendering.
12 pub mod toc;
13 /// Directory tree visualization.
14 pub mod tree;
15 /// GitHub user activity feed rendering.
16 pub mod user_activity;
17 /// User report cover page rendering.
18 pub mod user_cover;
19 /// User repository list rendering.
20 pub mod user_repos;
21
22 use std::path::Path;
23
24 use printpdf::{Mm, PdfDocument, PdfSaveOptions};
25
26 use crate::types::{Config, PaperSize, UserReportConfig};
27 use layout::{FontSet, PageBuilder};
28
29 fn paper_dimensions(config: &Config) -> (Mm, Mm) {
30     let (w, h) = match config.paper_size {
31         PaperSize::A4 => (Mm(210.0), Mm(297.0)),
32         PaperSize::Letter => (Mm(215.9), Mm(279.4)),
33         PaperSize::Legal => (Mm(215.9), Mm(355.6)),
34     };
35     if config.landscape { (h, w) } else { (w, h) }
36 }
37
38 /// Creates a `PageBuilder` starting at page 1 for the given config and font set.
39 pub fn create_builder(config: &Config, fonts: FontSet) -> PageBuilder {
40     create_builder_at_page(config, fonts, 1)
41 }
42
43 /// Creates a `PageBuilder` starting at an arbitrary page number (used to continue page numbering).
44 pub fn create_builder_at_page(
45     config: &Config,
46     fonts: FontSet,
47     starting_page: usize,
48 ) -> PageBuilder {
49     let (w, h) = paper_dimensions(config);
50     let line_height = config.font_size as f32 + 2.0;
51     PageBuilder::new(w, h, Mm(10.0), line_height, fonts, starting_page)
52 }
53
54 /// Creates a `PageBuilder` for a user report starting at page 1.
55 pub fn create_user_builder(config: &UserReportConfig, fonts: FontSet) -> PageBuilder {
56     create_user_builder_at_page(config, fonts, 1)
57 }
58
59 /// Creates a `PageBuilder` for a user report starting at an arbitrary page number.
60 pub fn create_user_builder_at_page(
61     config: &UserReportConfig,
62     fonts: FontSet,
63     starting_page: usize,
64 ) -> PageBuilder {
65     let (w, h) = match config.paper_size {
66         PaperSize::A4 => (Mm(210.0), Mm(297.0)),
67         PaperSize::Letter => (Mm(215.9), Mm(279.4)),
68         PaperSize::Legal => (Mm(215.9), Mm(355.6)),
69     };
70     let (w, h) = if config.landscape { (h, w) } else { (w, h) };
71     let line_height = config.font_size as f32 + 2.0;
72     PageBuilder::new(w, h, Mm(10.0), line_height, fonts, starting_page)
73 }
74
75 /// Serializes a `PdfDocument` to bytes and writes it to `path` asynchronously.
76 pub async fn save_pdf(doc: &PdfDocument, path: &Path) -> anyhow::Result<> {
77     let mut warnings = Vec::new();

```

```
78     let bytes = doc.save(&PdfSaveOptions::default(), &mut warnings);
79     tokio::fs::write(path, bytes).await.map_err(Into::into)
80 }
81
82 #[cfg(test)]
83 mod tests {
84     use super::*;
85     use crate::types::Config;
86
87     #[test]
88     fn paper_dimensions_a4() {
89         let config = Config::test_default();
90         let (w, h) = paper_dimensions(&config);
91         assert_eq!(w.0, 210.0);
92         assert_eq!(h.0, 297.0);
93     }
94
95     #[test]
96     fn paper_dimensions_letter() {
97         let mut config = Config::test_default();
98         config.paper_size = PaperSize::Letter;
99         let (w, h) = paper_dimensions(&config);
100        assert_eq!(w.0, 215.9);
101        assert_eq!(h.0, 279.4);
102    }
103
104    #[test]
105    fn paper_dimensions_landscape() {
106        let mut config = Config::test_default();
107        config.landscape = true;
108        let (w, h) = paper_dimensions(&config);
109        assert_eq!(w.0, 297.0);
110        assert_eq!(h.0, 210.0);
111    }
112
113    #[tokio::test]
114    async fn save_pdf_to_tempfile() {
115        let mut doc = PdfDocument::new("test");
116        let fonts = fonts::load_fonts(&mut doc).unwrap();
117        let config = Config::test_default();
118        let builder = create_builder(&config, fonts);
119        doc.with_pages(builder.finish());
120
121        let dir = tempfile::tempdir().unwrap();
122        let path = dir.path().join("test.pdf");
123        assert!(save_pdf(&doc, &path).await.is_ok());
124        assert!(path.exists());
125        assert!(std::fs::metadata(&path).unwrap().len() > 0);
126    }
127
128    #[tokio::test]
129    async fn save_pdf_invalid_path() {
130        let mut doc = PdfDocument::new("test");
131        let _ = fonts::load_fonts(&mut doc).unwrap();
132        let result = save_pdf(&doc, Path::new("/nonexistent/dir/test.pdf")).await;
133        assert!(result.is_err());
134    }
135 }
```

## src/pdf/toc.rs

```

1 use std::path::PathBuf;
2
3 use printpdf::{Actions, Color, Destination, Pt, Rgb};
4
5 use super::layout::{PageBuilder, Span};
6
7 /// A single entry in the Table of Contents.
8 pub struct TocEntry {
9     /// Path to the file relative to the repository root.
10    pub path: PathBuf,
11    /// Number of lines in the file.
12    pub line_count: usize,
13    /// Human-readable file size (e.g. "4.2 KB").
14    pub size_str: String,
15    /// Date the file was last modified (YYYY-MM-DD).
16    pub last_modified: String,
17    /// PDF page number where this file's content begins.
18    pub start_page: usize,
19 }
20
21 /// Split `text` into chunks of at most `max_chars` characters each.
22 fn wrap_text(text: &str, max_chars: usize) -> Vec<String> {
23     if max_chars == 0 || text.is_empty() {
24         return vec![text.to_string()];
25     }
26     let mut chunks = Vec::new();
27     let mut remaining = text;
28     while !remaining.is_empty() {
29         let split_at = remaining
30             .char_indices()
31             .nth(max_chars)
32             .map(|(i, _)| i)
33             .unwrap_or(remaining.len());
34         chunks.push(remaining[..split_at].to_string());
35         remaining = &remaining[split_at..];
36     }
37     chunks
38 }
39
40 /// Renders the table of contents page with clickable internal links for each entry.
41 pub fn render(builder: &mut PageBuilder, entries: &[TocEntry]) {
42     let bold = builder.font(true, false).clone();
43     let regular = builder.font(false, false).clone();
44     let gray = Color::Rgb(Rgb::new(0.47, 0.47, 0.47, None));
45     let black = Color::Rgb(Rgb::new(0.0, 0.0, 0.0, None));
46
47     builder.write_centered("Table of Contents", &bold, Pt(16.0), black);
48     builder.vertical_space(10.0);
49
50     // Approximate character width factors (monospace font approximation).
51     const PATH_SIZE: f32 = 8.0;
52     const META_SIZE: f32 = 7.0;
53     const CHAR_WIDTH: f32 = 0.6;
54     const GAP_PT: f32 = 8.0;
55
56     entries.iter().for_each(|entry| {
57         let meta = format!(
58             "p.{} {} LOC \u{00B7} {} \u{00B7} {}",
59             entry.start_page, entry.line_count, entry.size_str, entry.last_modified
60         );
61         let meta_width = meta.len() as f32 * META_SIZE * CHAR_WIDTH;
62         let available_left = builder.usable_width_pt() - meta_width - GAP_PT;
63         let max_chars = (available_left / (PATH_SIZE * CHAR_WIDTH)).max(1.0) as usize;
64
65         let path_str = entry.path.display().to_string();
66         let chunks = wrap_text(&path_str, max_chars);
67         let row_count = chunks.len();
68
69         // First chunk shares the line with meta; remaining chunks are on their own lines.
70         builder.write_line_justified(
71             &[Span {
72                 text: chunks[0].clone(),
73                 font_id: regular.clone(),
74                 size: Pt(PATH_SIZE),
75                 color: gray.clone(),
76             }],
77             &[Span {

```

```
78         text: meta,
79         font_id: regular.clone(),
80         size: Pt(META_SIZE),
81         color: gray.clone(),
82     }],
83 );
84 chunks[1..].iter().for_each(|chunk| {
85     builder.write_line(&{Span {
86         text: chunk.clone(),
87         font_id: regular.clone(),
88         size: Pt(PATH_SIZE),
89         color: gray.clone(),
90     }});
91 });
92
93 builder.add_link(
94     builder.line_height() * row_count as f32,
95     Actions::Goto(Destination::Xyz {
96         page: entry.start_page,
97         left: None,
98         top: None,
99         zoom: None,
100     }),
101 );
102 });
103
104 builder.page_break();
105 }
106
107 #[cfg(test)]
108 mod tests {
109     use crate::pdf;
110     use crate::types::Config;
111     use std::path::PathBuf;
112
113     fn make_entry(path: &str, lines: usize, page: usize) -> super::TocEntry {
114         super::TocEntry {
115             path: PathBuf::from(path),
116             line_count: lines,
117             size_str: "1.2 KB".to_string(),
118             last_modified: "2024-01-15".to_string(),
119             start_page: page,
120         }
121     }
122
123     #[test]
124     fn wrap_text_short() {
125         let chunks = super::wrap_text("short", 20);
126         assert_eq!(chunks, vec!["short"]);
127     }
128
129     #[test]
130     fn wrap_text_exact() {
131         let chunks = super::wrap_text("1234567890", 10);
132         assert_eq!(chunks, vec!["1234567890"]);
133     }
134
135     #[test]
136     fn wrap_text_overflow() {
137         let chunks = super::wrap_text("1234567890ab", 10);
138         assert_eq!(chunks, vec!["1234567890", "ab"]);
139     }
140
141     #[test]
142     fn wrap_text_empty() {
143         let chunks = super::wrap_text("", 10);
144         assert_eq!(chunks, vec![""]);
145     }
146
147     #[test]
148     fn render_toc_does_not_panic() {
149         let mut doc = printpdf::PdfDocument::new("test");
150         let fonts = pdf::fonts::load_fonts(&mut doc).unwrap();
151         let config = Config::test_default();
152         let mut builder = pdf::create_builder(&config, fonts);
153         let entries = vec![
154             make_entry("src/main.rs", 20, 5),
155             make_entry("src/lib.rs", 50, 7),
```

```
156     ];
157     super::render(&mut builder, &entries);
158 }
159
160 #[test]
161 fn render_toc_empty_files() {
162     let mut doc = printpdf::PdfDocument::new("test");
163     let fonts = pdf::fonts::load_fonts(&mut doc).unwrap();
164     let config = Config::test_default();
165     let mut builder = pdf::create_builder(&config, fonts);
166     super::render(&mut builder, &[]);
167 }
168
169 #[test]
170 fn render_toc_many_files() {
171     let mut doc = printpdf::PdfDocument::new("test");
172     let fonts = pdf::fonts::load_fonts(&mut doc).unwrap();
173     let config = Config::test_default();
174     let mut builder = pdf::create_builder(&config, fonts);
175     let entries: Vec<_> = (0..100)
176         .map(|i| make_entry("src/file.rs", i * 10, i + 5))
177         .collect();
178     super::render(&mut builder, &entries);
179 }
180
181 #[test]
182 fn render_toc_long_path_does_not_panic() {
183     let mut doc = printpdf::PdfDocument::new("test");
184     let fonts = pdf::fonts::load_fonts(&mut doc).unwrap();
185     let config = Config::test_default();
186     let mut builder = pdf::create_builder(&config, fonts);
187     let entries = vec![make_entry(
188         "src/very/deeply/nested/path/that/is/quite/long/file.rs",
189         100,
190         3,
191     )];
192     super::render(&mut builder, &entries);
193 }
194 }
```

## src/pdf/tree.rs

```

1 use std::collections::BTreeMap;
2 use std::path::PathBuf;
3
4 use printpdf::{Color, Pt, Rgb};
5
6 use super::layout::PageBuilder;
7
8 /// A recursive directory tree. BTreeMap keeps entries sorted alphabetically.
9 struct Tree(BTreeMap<String, Tree>);
10
11 impl Tree {
12     fn new() -> Self {
13         Self(BTreeMap::new())
14     }
15
16     fn insert(&mut self, parts: &[&str]) {
17         if let [first, rest @ ..] = parts {
18             self.0
19                 .entry(first.to_string())
20                 .or_insert_with(Tree::new)
21                 .insert(rest);
22         }
23     }
24
25     fn to_lines(&self, prefix: &str) -> Vec<String> {
26         let last_idx = self.0.len().saturating_sub(1);
27
28         self.0
29             .iter()
30             .enumerate()
31             .flat_map(|(i, (name, child))| {
32                 let is_last = i == last_idx;
33                 let connector = if is_last {
34                     "\u{2514}\u{2500}\u{2500} "
35                 } else {
36                     "\u{251C}\u{2500}\u{2500} "
37                 };
38                 let extension = if is_last { " " } else { "\u{2502} " };
39
40                 std::iter::once(format!("{prefix}{connector}{name}"))
41                     .chain(child.to_lines(&format!("{prefix}{extension}")))
42             })
43             .collect()
44     }
45 }
46
47 /// Renders a directory tree page showing all included file paths in box-drawing style.
48 pub fn render(builder: &mut PageBuilder, paths: &[PathBuf]) {
49     let bold = builder.font(true, false).clone();
50     let regular = builder.font(false, false).clone();
51     let black = Color::Rgb(Rgb::new(0.0, 0.0, 0.0, None));
52
53     builder.write_centered("File Tree", &bold, Pt(16.0), black.clone());
54     builder.vertical_space(10.0);
55
56     let mut root = Tree::new();
57     paths.iter().for_each(|p| {
58         let parts: Vec<_> = p
59             .components()
60             .map(|c| c.as_os_str().to_str().unwrap_or("?"))
61             .collect();
62         root.insert(&parts);
63     });
64
65     root.to_lines("").into_iter().for_each(|line| {
66         builder.write_line(&[super::layout::Span {
67             text: line,
68             font_id: regular.clone(),
69             size: Pt(7.0),
70             color: black.clone(),
71         }]);
72     });
73
74     builder.page_break();
75 }
76
77 #[cfg(test)]

```

```
78 mod tests {
79     use super::*;
80
81     #[test]
82     fn single_file() {
83         let mut tree = Tree::new();
84         tree.insert(&["src", "main.rs"]);
85         let lines = tree.to_lines("");
86         assert_eq!(lines.len(), 2);
87         assert!(lines[0].contains("src"));
88         assert!(lines[1].contains("main.rs"));
89     }
90
91     #[test]
92     fn nested_structure_with_box_drawing() {
93         let mut tree = Tree::new();
94         tree.insert(&["src", "main.rs"]);
95         tree.insert(&["src", "lib.rs"]);
96         tree.insert(&["Cargo.toml"]);
97         let lines = tree.to_lines("");
98         assert!(lines.len() >= 4);
99         let joined = lines.join("\n");
100        assert!(joined.contains('\u{251C}'));
101        assert!(joined.contains('\u{2514}'));
102        assert!(joined.contains('\u{2500}'));
103    }
104
105    #[test]
106    fn empty_tree() {
107        assert!(Tree::new().to_lines("").is_empty());
108    }
109
110    #[test]
111    fn sorted_output() {
112        let mut tree = Tree::new();
113        tree.insert(&["z.rs"]);
114        tree.insert(&["a.rs"]);
115        tree.insert(&["m.rs"]);
116        let lines = tree.to_lines("");
117        assert!(lines[0].contains("a.rs"));
118        assert!(lines[1].contains("m.rs"));
119        assert!(lines[2].contains("z.rs"));
120    }
121
122    #[test]
123    fn deep_nesting() {
124        let mut tree = Tree::new();
125        tree.insert(&["a", "b", "c", "d", "e.txt"]);
126        let lines = tree.to_lines("");
127        assert_eq!(lines.len(), 5);
128    }
129
130    #[test]
131    fn multiple_files_same_directory() {
132        let mut tree = Tree::new();
133        tree.insert(&["src", "a.rs"]);
134        tree.insert(&["src", "b.rs"]);
135        tree.insert(&["src", "c.rs"]);
136        assert_eq!(tree.to_lines("").len(), 4);
137    }
138
139    #[test]
140    fn render_does_not_panic() {
141        let mut doc = printpdf::PdfDocument::new("test");
142        let fonts = crate::pdf::fonts::load_fonts(&mut doc).unwrap();
143        let config = crate::types::Config::test_default();
144        let mut builder = crate::pdf::create_builder(&config, fonts);
145        render(
146            &mut builder,
147            &[
148                PathBuf::from("src/main.rs"),
149                PathBuf::from("src/lib.rs"),
150                PathBuf::from("Cargo.toml"),
151            ],
152        );
153    }
154 }
```

## src/pdf/user\_activity.rs

```

1 use printpdf::{Actions, Color, Pt, Rgb};
2
3 use super::layout::{PageBuilder, Span};
4 use crate::github::GitHubEvent;
5
6 /// Renders the "Recent Activity" section, grouping events by date with icons and links.
7 pub fn render(
8     builder: &mut PageBuilder,
9     events: &[GitHubEvent],
10    commit_msgs: &std::collections::HashMap<String, String>,
11 ) {
12     if events.is_empty() {
13         return;
14     }
15
16     let bold = builder.font(true, false).clone();
17     let regular = builder.font(false, false).clone();
18     let italic = builder.font(false, true).clone();
19     let black = Color::Rgb(Rgb::new(0.0, 0.0, 0.0, None));
20     let gray = Color::Rgb(Rgb::new(0.50, 0.50, 0.50, None));
21     let dark_gray = Color::Rgb(Rgb::new(0.25, 0.25, 0.25, None));
22     let rule_gray = Color::Rgb(Rgb::new(0.82, 0.82, 0.82, None));
23
24     // — Section title —
25     builder.ensure_space(builder.line_height() * 3.0);
26     builder.write_centered("Recent Activity", &bold, Pt(16.0), black.clone());
27     builder.vertical_space(10.0);
28     builder.draw_horizontal_rule(rule_gray.clone(), 0.5);
29     builder.vertical_space(8.0);
30
31     // — Events grouped by date —
32     let mut last_date = String::new();
33     events.iter().for_each(|event| {
34         let date = event.created_at.get(..10).unwrap_or(&event.created_at);
35         if date != last_date {
36             if !last_date.is_empty() {
37                 // Thin rule between date groups for visual separation.
38                 builder.vertical_space(4.0);
39                 builder.draw_horizontal_rule(rule_gray.clone(), 0.3);
40                 builder.vertical_space(8.0);
41             } else {
42                 builder.vertical_space(2.0);
43             }
44             builder.ensure_space(builder.line_height() * 2.0);
45             builder.write_line(&[Span {
46                 text: date.to_string(),
47                 font_id: bold.clone(),
48                 size: Pt(9.5),
49                 color: dark_gray.clone(),
50             }]);
51             last_date = date.to_string();
52             builder.vertical_space(2.0);
53         }
54
55         let time = event.created_at.get(11..16).unwrap_or("");
56         let description = describe_event(event);
57         let icon = event_icon(event);
58
59         // Enrich push events that have no commit info in the payload (force push /
60         // rebase). Look up this event's HEAD SHA to get its specific commit message.
61         let (main, detail) = if event.kind == "PushEvent" && description.detail.is_empty() {
62             let sha = event.payload["head"].as_str().unwrap_or("");
63             if let Some(msg) = commit_msgs.get(sha) {
64                 let branch = event.payload["ref"]
65                     .as_str()
66                     .unwrap_or("")
67                     .trim_start_matches("refs/heads/");
68                 let enriched_main = format!("Pushed to {} ({})", event.repo.name, event.repo.name);
69                 (enriched_main, vec![format!("{}", msg)])
70             } else {
71                 (description.main, description.detail)
72             }
73         } else {
74             (description.main, description.detail)
75         };
76
77         let url = event_url(event);

```

```

78     builder.write_line(&[
79         Span {
80             text: format!("{icon} "),
81             font_id: bold.clone(),
82             size: Pt(8.0),
83             color: event_icon_color(event),
84         },
85         Span {
86             text: format!("{time} "),
87             font_id: regular.clone(),
88             size: Pt(7.5),
89             color: gray.clone(),
90         },
91         Span {
92             text: main,
93             font_id: regular.clone(),
94             size: Pt(8.5),
95             color: black.clone(),
96         },
97     ]);
98     if let Some(u) = &url {
99         builder.add_link(builder.line_height(), Actions::Uri(u.clone()));
100     }
101
102     // Detail lines (commit messages, PR diff stats, etc.) – also link to the event.
103     detail.iter().for_each(|detail_line| {
104         builder.write_line(&[
105             Span {
106                 text: " ".to_string(),
107                 font_id: regular.clone(),
108                 size: Pt(7.5),
109                 color: gray.clone(),
110             },
111             Span {
112                 text: detail_line.clone(),
113                 font_id: italic.clone(),
114                 size: Pt(7.5),
115                 color: gray.clone(),
116             },
117         ]);
118         if let Some(u) = &url {
119             builder.add_link(builder.line_height(), Actions::Uri(u.clone()));
120         }
121     });
122 });
123
124 builder.vertical_space(12.0);
125 builder.page_break();
126 }
127
128 // — Event decorators —————
129
130 /// Single-character icon using Geometric Shapes (U+25A0-U+25FF) – all present
131 /// in JetBrains Mono and rendered reliably across PDF viewers including PDF.js.
132 fn event_icon(event: &GitHubEvent) -> &'static str {
133     match event.kind.as_str() {
134         "PushEvent" => "\u{25B6}", // ▶ black right-pointing triangle
135         "PullRequestEvent" => "\u{25B2}", // ▲ black up-pointing triangle
136         "IssuesEvent" => "\u{25CF}", // ● black circle
137         "IssueCommentEvent" => "\u{25E6}", // ◦ white bullet
138         "PullRequestReviewEvent" | "PullRequestReviewCommentEvent" => "\u{25CB}", // ○ white circle
139         "ReleaseEvent" => "\u{25C6}", // ◆ black diamond
140         "ForkEvent" => "\u{25B7}", // ▷ white right-pointing triangle
141         "WatchEvent" => "\u{25C6}", // ◆ black diamond (star/highlight)
142         "CreateEvent" => "\u{25AA}", // ▪ black small square
143         "DeleteEvent" => "\u{25AB}", // ◻ white small square
144         _ => "\u{00B7}", // · middle dot
145     }
146 }
147
148 fn event_icon_color(event: &GitHubEvent) -> Color {
149     match event.kind.as_str() {
150         "PushEvent" => Color::Rgb(Rgb::new(0.27, 0.68, 0.96, None)), // blue
151         "PullRequestEvent" => Color::Rgb(Rgb::new(0.55, 0.36, 0.90, None)), // purple
152         "IssuesEvent" => Color::Rgb(Rgb::new(0.96, 0.55, 0.13, None)), // orange
153         "IssueCommentEvent" | "PullRequestReviewEvent" | "PullRequestReviewCommentEvent" => {
154             Color::Rgb(Rgb::new(0.50, 0.50, 0.50, None)) // gray
155         }
156     }

```

```

156     "ReleaseEvent" => Color::Rgb(Rgb::new(0.13, 0.78, 0.47, None)), // green
157     "WatchEvent" => Color::Rgb(Rgb::new(0.96, 0.80, 0.10, None)), // gold
158     "ForkEvent" => Color::Rgb(Rgb::new(0.27, 0.68, 0.96, None)), // blue
159     "CreateEvent" => Color::Rgb(Rgb::new(0.13, 0.78, 0.47, None)), // green
160     "DeleteEvent" => Color::Rgb(Rgb::new(0.78, 0.25, 0.25, None)), // red
161     _ => Color::Rgb(Rgb::new(0.50, 0.50, 0.50, None)),
162 }
163 }
164
165 struct EventDescription {
166     main: String,
167     detail: Vec<String>,
168 }
169
170 fn describe_event(event: &GitHubEvent) -> EventDescription {
171     let repo = &event.repo.name;
172     let p = &event.payload;
173
174     let (main, detail) = match event.kind.as_str() {
175         "PushEvent" => {
176             let branch = p["ref"]
177                 .as_str()
178                 .unwrap_or("")
179                 .trim_start_matches("refs/heads/");
180             let commits_arr = p["commits"].as_array();
181             let count = p["size"]
182                 .as_u64()
183                 .map(|n| n as usize)
184                 .filter(|&n| n > 0)
185                 .unwrap_or_else(|| commits_arr.map(|c| c.len()).unwrap_or(0));
186             let main = if count > 0 {
187                 let label = if count == 1 { "commit" } else { "commits" };
188                 format!("Pushed {count} {label} to {repo} ({branch})")
189             } else {
190                 format!("Pushed to {repo} ({branch})")
191             };
192             let detail: Vec<String> = commits_arr
193                 .into_iter()
194                 .flatten()
195                 .take(5)
196                 .filter_map(|c| c["message"].as_str())
197                 .map(|m| format!("  {}", m.lines().next().unwrap_or(m)))
198                 .collect();
199             (main, detail)
200         }
201         "PullRequestEvent" => {
202             let action = p["action"].as_str().unwrap_or("updated");
203             let merged =
204                 action == "closed" && p["pull_request"]["merged"].as_bool().unwrap_or(false);
205             let label = if merged { "merged" } else { action };
206             let title = p["pull_request"]["title"].as_str().unwrap_or("");
207             let number = p["pull_request"]["number"].as_u64().unwrap_or(0);
208             let detail = match (
209                 p["pull_request"]["additions"].as_u64(),
210                 p["pull_request"]["deletions"].as_u64(),
211                 p["pull_request"]["changed_files"].as_u64(),
212             ) {
213                 (Some(a), Some(d), Some(f)) => {
214                     let fword = if f == 1 { "file" } else { "files" };
215                     vec![format!("    +{a} \u{2212}{d} across {f} {fword}")]
216                 }
217                 _ => vec![],
218             };
219             (
220                 format!("{ PR #number}: {title} in {repo}", capitalise(label)),
221                 detail,
222             )
223         }
224         "IssuesEvent" => {
225             let action = p["action"].as_str().unwrap_or("updated");
226             let title = p["issue"]["title"].as_str().unwrap_or("");
227             let number = p["issue"]["number"].as_u64().unwrap_or(0);
228             (
229                 format!("{ issue #number}: {title} in {repo}", capitalise(action)),
230                 vec![],
231             )
232         }
233         "IssueCommentEvent" => {

```

```

234     let title = p["issue"]["title"].as_str().unwrap_or("");
235     let number = p["issue"]["number"].as_u64().unwrap_or(0);
236     (
237         format!("Commented on issue #{number}: {title} in {repo}"),
238         vec![],
239     )
240 }
241 "PullRequestReviewEvent" => {
242     let state = p["review"]["state"].as_str().unwrap_or("reviewed");
243     let number = p["pull_request"]["number"].as_u64().unwrap_or(0);
244     (
245         format!("#{} PR #{number} in {repo}", capitalise(state)),
246         vec![],
247     )
248 }
249 "PullRequestReviewCommentEvent" => {
250     let number = p["pull_request"]["number"].as_u64().unwrap_or(0);
251     (format!("Reviewed PR #{number} in {repo}"), vec![])
252 }
253 "CreateEvent" => {
254     let ref_type = p["ref_type"].as_str().unwrap_or("ref");
255     let ref_name = p["ref"].as_str().unwrap_or("");
256     if ref_name.is_empty() {
257         (format!("Created {ref_type} {repo}"), vec![])
258     } else {
259         (format!("Created {ref_type} '{ref_name}' in {repo}"), vec![])
260     }
261 }
262 "DeleteEvent" => {
263     let ref_type = p["ref_type"].as_str().unwrap_or("ref");
264     let ref_name = p["ref"].as_str().unwrap_or("");
265     (format!("Deleted {ref_type} '{ref_name}' in {repo}"), vec![])
266 }
267 "ForkEvent" => {
268     let forkee = p["forkee"]["full_name"].as_str().unwrap_or(repo);
269     (format!("Forked {repo} \u{2192} {forkee}"), vec![])
270 }
271 "WatchEvent" => (format!("Starred {repo}"), vec![]),
272 "ReleaseEvent" => {
273     let action = p["action"].as_str().unwrap_or("published");
274     let tag = p["release"]["tag_name"].as_str().unwrap_or("");
275     (
276         format!("#{} release {tag} in {repo}", capitalise(action)),
277         vec![],
278     )
279 }
280 "CommitCommentEvent" => (format!("Commented on a commit in {repo}"), vec![]),
281 "GoLlumEvent" => (format!("Updated wiki in {repo}"), vec![]),
282 "MemberEvent" => {
283     let action = p["action"].as_str().unwrap_or("updated");
284     let member = p["member"]["login"].as_str().unwrap_or("someone");
285     (
286         format!("#{} {member} as collaborator in {repo}", capitalise(action)),
287         vec![],
288     )
289 }
290 "PublicEvent" => (format!("Made {repo} public"), vec![]),
291 "SponsorshipEvent" => (format!("Sponsorship activity in {repo}"), vec![]),
292 other => (format!("{other} in {repo}"), vec![]),
293 };
294
295 EventDescription { main, detail }
296 }
297
298 /// Returns the most relevant clickable URL for a Github event, if one exists.
299 fn event_url(event: &GitHubEvent) -> Option<String> {
300     let repo = &event.repo.name;
301     let p = &event.payload;
302     match event.kind.as_str() {
303         "PushEvent" => {
304             if let Some(sha) = p["head"].as_str() {
305                 Some(format!("https://github.com/{repo}/commit/{sha}"))
306             } else {
307                 p["ref"].as_str().map(|r| {
308                     let branch = r.trim_start_matches("refs/heads/");
309                     format!("https://github.com/{repo}/tree/{branch}")
310                 })
311             }
312         }

```

```

312     }
313     "PullRequestEvent" => p["pull_request"]["html_url"].as_str().map(str::to_string),
314     "IssuesEvent" => p["issue"]["html_url"].as_str().map(str::to_string),
315     "IssueCommentEvent" => p["comment"]["html_url"].as_str().map(str::to_string),
316     "PullRequestReviewEvent" | "PullRequestReviewCommentEvent" => {
317         p["pull_request"]["html_url"].as_str().map(str::to_string)
318     }
319     "ForkEvent" => p["forkee"]["html_url"].as_str().map(str::to_string),
320     "ReleaseEvent" => p["release"]["html_url"].as_str().map(str::to_string),
321     _ => Some(format!("https://github.com/{repo}")),
322 }
323 }
324
325 fn capitalise(s: &str) -> String {
326     let mut chars = s.chars();
327     match chars.next() {
328         None => String::new(),
329         Some(c) => c.to_uppercase().to_string() + chars.as_str(),
330     }
331 }
332
333 #[cfg(test)]
334 mod tests {
335     use super::*;
336     use crate::pdf;
337     use crate::types::Config;
338
339     fn push_event() -> GitHubEvent {
340         GitHubEvent {
341             kind: "PushEvent".to_string(),
342             repo: crate::github::EventRepo {
343                 name: "alice/myrepo".to_string(),
344             },
345             payload: serde_json::json!({
346                 "ref": "refs/heads/main",
347                 "commits": [
348                     { "message": "fix: correct typo" },
349                     { "message": "feat: add feature" }
350                 ]
351             }),
352             created_at: "2024-03-01T12:00:00Z".to_string(),
353         }
354     }
355
356     fn pr_event() -> GitHubEvent {
357         GitHubEvent {
358             kind: "PullRequestEvent".to_string(),
359             repo: crate::github::EventRepo {
360                 name: "alice/myrepo".to_string(),
361             },
362             payload: serde_json::json!({
363                 "action": "opened",
364                 "pull_request": { "number": 42, "title": "Add dark mode" }
365             }),
366             created_at: "2024-03-01T11:00:00Z".to_string(),
367         }
368     }
369
370     #[test]
371     fn render_activity_does_not_panic() {
372         let mut doc = printpdf::PdfDocument::new("test");
373         let fonts = pdf::fonts::load_fonts(&mut doc).unwrap();
374         let config = Config::test_default();
375         let mut builder = pdf::create_builder(&config, fonts);
376         super::render(
377             &mut builder,
378             &[push_event(), pr_event()],
379             &std::collections::HashMap::new(),
380         );
381         assert(!builder.finish().is_empty());
382     }
383
384     #[test]
385     fn render_activity_empty_is_noop() {
386         let mut doc = printpdf::PdfDocument::new("test");
387         let fonts = pdf::fonts::load_fonts(&mut doc).unwrap();
388         let config = Config::test_default();
389         let mut builder = pdf::create_builder(&config, fonts);

```

```

390     let page_before = builder.current_page();
391     super::render(&mut builder, &[], &std::collections::HashMap::new());
392     assert_eq!(builder.current_page(), page_before);
393 }
394
395 #[test]
396 fn capitalise_works() {
397     assert_eq!(super::capitalise("opened"), "Opened");
398     assert_eq!(super::capitalise(""), "");
399     assert_eq!(super::capitalise("x"), "X");
400 }
401
402 #[test]
403 fn describe_push_event() {
404     let desc = super::describe_event(&push_event());
405     assert!(desc.main.contains("Pushed 2 commits"));
406     assert!(desc.main.contains("alice/myrepo"));
407     assert!(desc.main.contains("main"));
408     assert_eq!(desc.detail.len(), 2);
409 }
410
411 #[test]
412 fn describe_pr_event() {
413     let desc = super::describe_event(&pr_event());
414     assert!(desc.main.contains("Opened PR #42"));
415     assert!(desc.main.contains("Add dark mode"));
416 }
417
418 #[test]
419 fn event_icons_cover_all_known_types() {
420     [
421         "PushEvent",
422         "PullRequestEvent",
423         "IssuesEvent",
424         "ReleaseEvent",
425         "WatchEvent",
426         "IssueCommentEvent",
427         "PullRequestReviewEvent",
428         "PullRequestReviewCommentEvent",
429         "ForkEvent",
430         "CreateEvent",
431         "DeleteEvent",
432         "UnknownEvent",
433     ]
434     .iter()
435     .for_each(|kind| {
436         let e = GitHubEvent {
437             kind: kind.to_string(),
438             repo: crate::github::EventRepo {
439                 name: "a/b".to_string(),
440             },
441             payload: serde_json::json!({}),
442             created_at: "2024-01-01T00:00:00Z".to_string(),
443         };
444         assert!(!super::event_icon(&e).is_empty());
445         // icon color must not panic
446         let _ = super::event_icon_color(&e);
447     });
448 }
449
450 fn make_event(kind: &str, payload: serde_json::Value) -> GitHubEvent {
451     GitHubEvent {
452         kind: kind.to_string(),
453         repo: crate::github::EventRepo {
454             name: "alice/repo".to_string(),
455         },
456         payload,
457         created_at: "2024-03-01T09:30:00Z".to_string(),
458     }
459 }
460
461 #[test]
462 fn describe_issue_comment_event() {
463     let e = make_event(
464         "IssueCommentEvent",
465         serde_json::json!({ "issue": { "number": 7, "title": "Bug" } }),
466     );
467     let d = super::describe_event(&e);

```

```
468     assert!(d.main.contains("#7"));
469     assert!(d.main.contains("Bug"));
470 }
471
472 #[test]
473 fn describe_pr_review_event() {
474     let e = make_event(
475         "PullRequestReviewEvent",
476         serde_json::json!({ "review": { "state": "approved" }, "pull_request": { "number": 3 } })),
477     );
478     let d = super::describe_event(&e);
479     assert!(d.main.contains("Approved"));
480     assert!(d.main.contains("#3"));
481 }
482
483 #[test]
484 fn describe_pr_review_comment_event() {
485     let e = make_event(
486         "PullRequestReviewCommentEvent",
487         serde_json::json!({ "pull_request": { "number": 5 } })),
488     );
489     let d = super::describe_event(&e);
490     assert!(d.main.contains("#5"));
491 }
492
493 #[test]
494 fn describe_create_event_with_ref() {
495     let e = make_event(
496         "CreateEvent",
497         serde_json::json!({ "ref_type": "branch", "ref": "feature/x" })),
498     );
499     let d = super::describe_event(&e);
500     assert!(d.main.contains("branch"));
501     assert!(d.main.contains("feature/x"));
502 }
503
504 #[test]
505 fn describe_create_event_no_ref() {
506     let e = make_event(
507         "CreateEvent",
508         serde_json::json!({ "ref_type": "repository", "ref": "" })),
509     );
510     let d = super::describe_event(&e);
511     assert!(d.main.contains("repository"));
512     assert!(d.main.contains(""));
513 }
514
515 #[test]
516 fn describe_delete_event() {
517     let e = make_event(
518         "DeleteEvent",
519         serde_json::json!({ "ref_type": "branch", "ref": "old-feature" })),
520     );
521     let d = super::describe_event(&e);
522     assert!(d.main.contains("old-feature"));
523 }
524
525 #[test]
526 fn describe_fork_event() {
527     let e = make_event(
528         "ForkEvent",
529         serde_json::json!({ "forkee": { "full_name": "bob/repo" } })),
530     );
531     let d = super::describe_event(&e);
532     assert!(d.main.contains("bob/repo"));
533 }
534
535 #[test]
536 fn describe_watch_event() {
537     let d = super::describe_event(&make_event("WatchEvent", serde_json::json!({})));
538     assert!(d.main.contains("Starred"));
539 }
540
541 #[test]
542 fn describe_release_event() {
543     let e = make_event(
544         "ReleaseEvent",
545         serde_json::json!({ "action": "published", "release": { "tag_name": "v1.2.3" } })),
```

```

546     );
547     let d = super::describe_event(&e);
548     assert!(d.main.contains("v1.2.3"));
549 }
550
551 #[test]
552 fn describe_commit_comment_event() {
553     let d = super::describe_event(&make_event("CommitCommentEvent", serde_json::json!({})));
554     assert!(d.main.contains("commit"));
555 }
556
557 #[test]
558 fn describe_gollum_event() {
559     let d = super::describe_event(&make_event("GollumEvent", serde_json::json!({})));
560     assert!(d.main.contains("wiki"));
561 }
562
563 #[test]
564 fn describe_member_event() {
565     let e = make_event(
566         "MemberEvent",
567         serde_json::json!({ "action": "added", "member": { "login": "bob" } })),
568     );
569     let d = super::describe_event(&e);
570     assert!(d.main.contains("bob"));
571 }
572
573 #[test]
574 fn describe_public_event() {
575     let d = super::describe_event(&make_event("PublicEvent", serde_json::json!({})));
576     assert!(d.main.contains("public"));
577 }
578
579 #[test]
580 fn describe_unknown_event() {
581     let d = super::describe_event(&make_event("CoolNewEvent", serde_json::json!({})));
582     assert!(!d.main.is_empty());
583 }
584
585 #[test]
586 fn describe_pr_event_merged() {
587     let e = make_event(
588         "PullRequestEvent",
589         serde_json::json!({
590             "action": "closed",
591             "pull_request": {
592                 "number": 10, "title": "Big feature",
593                 "merged": true,
594                 "additions": 50, "deletions": 5, "changed_files": 3
595             }
596         })),
597     );
598     let d = super::describe_event(&e);
599     assert!(d.main.contains("Merged"));
600     assert_eq!(d.detail.len(), 1);
601 }
602
603 #[test]
604 fn describe_push_event_no_size_field() {
605     // When "size" is absent, falls back to commits array length.
606     let e = make_event(
607         "PushEvent",
608         serde_json::json!({
609             "ref": "refs/heads/main",
610             "commits": [{ "message": "only commit" }]
611         })),
612     );
613     let d = super::describe_event(&e);
614     assert!(d.main.contains("1 commit"));
615 }
616
617 #[test]
618 fn event_url_push_without_head_uses_branch() {
619     let e = make_event("PushEvent", serde_json::json!({ "ref": "refs/heads/feat" }));
620     let url = super::event_url(&e).unwrap();
621     assert!(url.contains("feat"));
622 }
623

```

```
624 #[test]
625 fn event_url_pull_request_uses_html_url() {
626     let e = make_event(
627         "PullRequestEvent",
628         serde_json::json!({ "pull_request": { "html_url": "https://github.com/alice/repo/pull/1" } })),
629     );
630     assert_eq!(
631         super::event_url(&e),
632         Some("https://github.com/alice/repo/pull/1".to_string())
633     );
634 }
635
636 #[test]
637 fn event_url_issues_event() {
638     let e = make_event(
639         "IssuesEvent",
640         serde_json::json!({ "issue": { "html_url": "https://github.com/alice/repo/issues/2" } })),
641     );
642     assert_eq!(
643         super::event_url(&e),
644         Some("https://github.com/alice/repo/issues/2".to_string())
645     );
646 }
647
648 #[test]
649 fn event_url_fork_event() {
650     let e = make_event(
651         "ForkEvent",
652         serde_json::json!({ "forkee": { "html_url": "https://github.com/bob/repo" } })),
653     );
654     assert_eq!(
655         super::event_url(&e),
656         Some("https://github.com/bob/repo".to_string())
657     );
658 }
659
660 #[test]
661 fn event_url_catchall_returns_repo_url() {
662     let url = super::event_url(&make_event("WatchEvent", serde_json::json!({}))).unwrap();
663     assert!(url.contains("alice/repo"));
664 }
665
666 #[test]
667 fn render_activity_many_event_types() {
668     let mut doc = printpdf::PdfDocument::new("test");
669     let fonts = pdf::fonts::load_fonts(&mut doc).unwrap();
670     let config = Config::test_default();
671     let mut builder = pdf::create_builder(&config, fonts);
672     let events: Vec<GitHubEvent> = [
673         "IssuesEvent",
674         "IssueCommentEvent",
675         "PullRequestReviewEvent",
676         "CreateEvent",
677         "DeleteEvent",
678         "ForkEvent",
679         "ReleaseEvent",
680         "WatchEvent",
681     ]
682     .iter()
683     .map(|kind| make_event(kind, serde_json::json!({})))
684     .collect();
685     super::render(&mut builder, &events, &std::collections::HashMap::new());
686     assert!(builder.finish().is_empty());
687 }
688 }
```

## src/pdf/user\_cover.rs

```

1 use printpdf::{Actions, Color, Pt, Rgb};
2
3 use super::layout::{PageBuilder, Span};
4 use crate::github::GitHubUser;
5
6 const CRATES_URL: &str = "https://crates.io/crates/gitprint";
7 const LABEL_COL: usize = 14;
8 const CHAR_WIDTH: f32 = 0.6;
9
10 fn separator_line(width_pt: f32, font_size: f32) -> String {
11     let chars = (width_pt / (font_size * CHAR_WIDTH)).max(1.0) as usize;
12     "-".repeat(chars)
13 }
14
15 /// Word-wrap `text` into lines of at most `max_chars` characters, breaking at word boundaries.
16 fn word_wrap(text: &str, max_chars: usize) -> Vec<String> {
17     if max_chars == 0 {
18         return vec![text.to_string()];
19     }
20     let (mut lines, last) = text.split_whitespace().fold(
21         (Vec::<String>::new(), String::new()),
22         |(mut lines, mut cur), word| {
23             if !cur.is_empty() && cur.len() + 1 + word.len() > max_chars {
24                 lines.push(std::mem::take(&mut cur));
25             } else if !cur.is_empty() {
26                 cur.push(' ');
27             }
28             cur.push_str(word);
29             (lines, cur)
30         },
31     );
32     if !last.is_empty() {
33         lines.push(last);
34     }
35     lines
36 }
37
38 /// Renders the user report cover page with profile info, metadata table, and footer.
39 pub fn render(builder: &mut PageBuilder, user: &GitHubUser, total_stars: u64) {
40     let bold = builder.font(true, false).clone();
41     let regular = builder.font(false, false).clone();
42     let black = Color::Rgb(Rgb::new(0.0, 0.0, 0.0, None));
43     let gray = Color::Rgb(Rgb::new(0.47, 0.47, 0.47, None));
44     let lh = builder.line_height();
45
46     const TABLE_SIZE: f32 = 9.0;
47
48     let display_name = user.name.as_deref().unwrap_or(&user.login);
49
50     // — Title —————
51     builder.vertical_space(120.0);
52     builder.write_centered(display_name, &bold, Pt(28.0), black.clone());
53     builder.add_link(28.0 + 4.0, Actions::Uri(user.html_url.clone()));
54
55     if display_name != user.login {
56         builder.vertical_space(6.0);
57         builder.write_centered(
58             &format!("{}", user.login),
59             &regular,
60             Pt(12.0),
61             gray.clone(),
62         );
63     }
64     builder.add_link(12.0 + 4.0, Actions::Uri(user.html_url.clone()));
65
66     builder.vertical_space(32.0);
67
68     // — Metadata table —————
69     builder.draw_horizontal_rule(Color::Rgb(Rgb::new(0.75, 0.75, 0.75, None)), 0.5);
70     builder.vertical_space(8.0);
71
72     // Build rows dynamically — only show non-empty fields.
73     let repos_str = user.public_repos.to_string();
74     let stars_str = total_stars.to_string();
75     let followers_str = user.followers.to_string();
76     let following_str = user.following.to_string();
77     let member_since = user

```

```

78     .created_at
79     .get(..10)
80     .unwrap_or(&user.created_at)
81     .to_string();
82
83     let value_col_max_chars = ((builder.usable_width_pt()
84     - LABEL_COL as f32 * TABLE_SIZE * CHAR_WIDTH)
85     / (TABLE_SIZE * CHAR_WIDTH))
86     .max(1.0) as usize;
87
88     let email_url = user.email.as_ref().map(|e| format!("mailto:{e}"));
89     let repos_url = format!("{:?}tab=repositories", user.html_url);
90     let followers_url = format!("{:?}tab=followers", user.html_url);
91     let following_url = format!("{:?}tab=following", user.html_url);
92
93     [
94     ("Bio", user.bio.as_deref().unwrap_or(""), None::<String>),
95     ("Location", user.location.as_deref().unwrap_or(""), None),
96     ("Company", user.company.as_deref().unwrap_or(""), None),
97     (
98     "Blog",
99     user.blog.as_deref().unwrap_or(""),
100    user.blog.as_ref().map(|b| {
101        if b.starts_with("http") {
102            b.clone()
103        } else {
104            format!("https://{b}")
105        }
106    }),
107    ),
108    (
109    "Email",
110    user.email.as_deref().unwrap_or(""),
111    email_url.clone(),
112    ),
113    ("Public Repos", &repos_str, Some(repos_url.clone())),
114    ("Total Stars", &stars_str, None),
115    ("Followers", &followers_str, Some(followers_url.clone())),
116    ("Following", &following_str, Some(following_url.clone())),
117    ("Member Since", &member_since, None),
118    ("Profile", &user.html_url, Some(user.html_url.clone())),
119    ]
120    .into_iter()
121    .filter(|(_, value, _)| !value.is_empty())
122    .for_each(|(label, value, url)| {
123        word_wrap(value, value_col_max_chars)
124            .into_iter()
125            .enumerate()
126            .for_each(|(i, line)| {
127                let label_text = if i == 0 {
128                    format!("{label:<LABEL_COL$}")
129                } else {
130                    " ".repeat(LABEL_COL)
131                };
132                builder.write_line(&[
133                    Span {
134                        text: label_text,
135                        font_id: bold.clone(),
136                        size: Pt(TABLE_SIZE),
137                        color: black.clone(),
138                    },
139                    Span {
140                        text: line,
141                        font_id: regular.clone(),
142                        size: Pt(TABLE_SIZE),
143                        color: black.clone(),
144                    },
145                ]);
146            });
147        if let Some(u) = url {
148            builder.add_link(lh, Actions::Uri(u));
149        }
150    });
151
152    builder.vertical_space(4.0);
153    builder.draw_horizontal_rule(Color::Rgb(Rgb::new(0.75, 0.75, 0.75, None)), 0.5);
154
155    // — Footer —

```

```

156 let version = env!("CARGO_PKG_VERSION");
157 let footer_text =
158     format!("Generated with gitprint v{version} ({CRATES_URL}), a Izel Nakri production");
159 let footer_size = Pt(7.0);
160 let footer_area = lh + 4.0 + footer_size.0 + 4.0;
161 builder.vertical_space((builder.remaining_pt() - footer_area).max(0.0));
162
163 builder.write_line(&[Span {
164     text: separator_line(builder.usable_width_pt(), footer_size.0),
165     font_id: regular.clone(),
166     size: footer_size,
167     color: gray.clone(),
168 }]);
169 builder.vertical_space(4.0);
170 builder.write_centered(&footer_text, &regular, footer_size, gray);
171 builder.add_link(footer_size.0 + 4.0, Actions::Uri(CRATES_URL.to_string()));
172
173 builder.page_break();
174 }
175
176 #[cfg(test)]
177 mod tests {
178     use super::*;
179     use crate::pdf;
180     use crate::types::{Config, UserReportConfig};
181
182     fn test_user() -> GitHubUser {
183         GitHubUser {
184             login: "alice".to_string(),
185             name: Some("Alice Dev".to_string()),
186             bio: Some("Rust enthusiast".to_string()),
187             location: Some("Berlin".to_string()),
188             company: Some("Acme Corp".to_string()),
189             blog: Some("https://alice.dev".to_string()),
190             email: Some("alice@example.com".to_string()),
191             public_repos: 42,
192             followers: 100,
193             following: 50,
194             created_at: "2018-03-15T10:00:00Z".to_string(),
195             html_url: "https://github.com/alice".to_string(),
196         }
197     }
198
199     fn test_user_config() -> UserReportConfig {
200         UserReportConfig {
201             username: "alice".to_string(),
202             output_path: "/tmp/alice.pdf".into(),
203             paper_size: crate::types::PaperSize::A4,
204             landscape: false,
205             last_repos: 5,
206             last_commits: 5,
207             no_diffs: false,
208             font_size: 8.0,
209             github_token: None,
210             since: None,
211             until: None,
212             activity: crate::types::ActivityFilter::All,
213             events: 30,
214         }
215     }
216
217     #[test]
218     fn render_user_cover_does_not_panic() {
219         let mut doc = printpdf::PdfDocument::new("test");
220         let fonts = pdf::fonts::load_fonts(&mut doc).unwrap();
221         let config = Config::test_default();
222         let mut builder = pdf::create_builder(&config, fonts);
223         super::render(&mut builder, &test_user(), 1337);
224         assert!(!builder.finish().is_empty());
225     }
226
227     #[test]
228     fn render_user_cover_no_name() {
229         let mut doc = printpdf::PdfDocument::new("test");
230         let fonts = pdf::fonts::load_fonts(&mut doc).unwrap();
231         let config = Config::test_default();
232         let mut builder = pdf::create_builder(&config, fonts);
233         let mut user = test_user();

```

```
234     user.name = None;
235     super::render(&mut builder, &user, 0);
236     assert(!builder.finish().is_empty());
237 }
238
239 #[test]
240 fn render_user_cover_minimal_fields() {
241     let mut doc = printpdf::PdfDocument::new("test");
242     let fonts = pdf::fonts::load_fonts(&mut doc).unwrap();
243     let uc = test_user_config();
244     let mut builder = pdf::create_user_builder(&uc, fonts);
245     let user = GitHubUser {
246         login: "bob".to_string(),
247         name: None,
248         bio: None,
249         location: None,
250         company: None,
251         blog: None,
252         email: None,
253         public_repos: 1,
254         followers: 0,
255         following: 0,
256         created_at: "2020-01-01T00:00:00Z".to_string(),
257         html_url: "https://github.com/bob".to_string(),
258     };
259     super::render(&mut builder, &user, 0);
260     assert(!builder.finish().is_empty());
261 }
262 }
```

## src/pdf/user\_repos.rs

```

1 use std::collections::HashMap;
2
3 use printpdf::{Actions, Color, Pt, Rgb};
4
5 use super::layout::{PageBuilder, Span};
6 use crate::github::{GitHubEvent, GitHubRepo};
7
8 /// Renders a titled section listing repositories with stats and recent activity context.
9 pub fn render(
10     builder: &mut PageBuilder,
11     title: &str,
12     repos: &[GitHubRepo],
13     events: &[GitHubEvent],
14     commit_msgs: &std::collections::HashMap<String, String>,
15 ) {
16     if repos.is_empty() {
17         return;
18     }
19
20     // Index events by repo name, keeping the newest (API returns newest-first).
21     let push_ctx: HashMap<&str, &GitHubEvent> = events
22         .iter()
23         .filter(|e| e.kind == "PushEvent")
24         .fold(HashMap::new(), |mut map, e| {
25             map.entry(e.repo.name.as_str()).or_insert(e);
26             map
27         });
28     let activity_ctx: HashMap<&str, &GitHubEvent> = events
29         .iter()
30         .filter(|e| e.kind != "PushEvent")
31         .fold(HashMap::new(), |mut map, e| {
32             map.entry(e.repo.name.as_str()).or_insert(e);
33             map
34         });
35
36     let bold = builder.font(true, false).clone();
37     let regular = builder.font(false, false).clone();
38     let italic = builder.font(false, true).clone();
39     let black = Color::Rgb(Rgb::new(0.0, 0.0, 0.0, None));
40     let gray = Color::Rgb(Rgb::new(0.47, 0.47, 0.47, None));
41     let dark_gray = Color::Rgb(Rgb::new(0.25, 0.25, 0.25, None));
42     let gold = Color::Rgb(Rgb::new(0.90, 0.72, 0.10, None));
43     let rule_gray = Color::Rgb(Rgb::new(0.85, 0.85, 0.85, None));
44
45     builder.ensure_space(builder.line_height() * 3.0);
46     builder.write_centered(title, &bold, Pt(14.0), black.clone());
47     builder.vertical_space(8.0);
48     builder.draw_horizontal_rule(rule_gray.clone(), 0.5);
49     builder.vertical_space(8.0);
50
51     repos.iter().enumerate().for_each(|(idx, repo)| {
52         // Thin separator between repo entries (not before the first one).
53         if idx > 0 {
54             builder.vertical_space(2.0);
55             builder.draw_horizontal_rule(rule_gray.clone(), 0.3);
56             builder.vertical_space(8.0);
57         }
58
59         builder.ensure_space(builder.line_height() * 5.0);
60
61         // — Row 1: name (left) + stats (right) —————
62         let fork_tag = if repo.fork { "[fork]" } else { "" };
63         let lang = repo.language.as_deref().unwrap_or("-");
64         let stats = format!(
65             "\u{25C6} {} \u{00B7} \u{25B7} {} \u{00B7} ! {} \u{00B7} {}",
66             repo.stargazers_count, repo.forks_count, repo.open_issues_count, lang
67         );
68         builder.write_line_justified(
69             &[Span {
70                 text: format!("{}{}", fork_tag, repo.name),
71                 font_id: bold.clone(),
72                 size: Pt(9.0),
73                 color: black.clone(),
74             }],
75             &[Span {
76                 text: stats,
77                 font_id: regular.clone(),

```

```

78         size: Pt(8.0),
79         color: gold.clone(),
80     }],
81 );
82 builder.add_link(builder.line_height(), Actions::Uri(repo.html_url.clone()));
83
84 // — Row 2: description —————
85 if let Some(desc) = repo.description.as_deref().filter(|d| !d.is_empty()) {
86     builder.write_line(&[Span {
87         text: format!(" {desc}"),
88         font_id: italic.clone(),
89         size: Pt(8.0),
90         color: gray.clone(),
91     }]);
92 }
93
94 // — Row 3: dates + size —————
95 let pushed = repo
96     .pushed_at
97     .as_deref()
98     .or(repo.updated_at.as_deref())
99     .and_then(|d| d.get(..10))
100    .unwrap_or("-");
101 let created = repo
102     .created_at
103     .as_deref()
104     .and_then(|d| d.get(..10))
105     .unwrap_or("-");
106 let size_part = match repo.size {
107     0 => String::new(),
108     kb if kb < 1024 => format!(" . {kb} KB"),
109     kb => format!(" · {:.1} MB", kb as f64 / 1024.0),
110 };
111 builder.write_line(&[Span {
112     text: format!(" last push {pushed} · created {created}{size_part}"),
113     font_id: regular.clone(),
114     size: Pt(7.5),
115     color: gray.clone(),
116 }]);
117
118 // — Row 4: your recent activity context —————
119 // Push event → show branch + commit messages (you pushed code here).
120 // Non-push event → show what you did (opened issue, reviewed PR, etc.).
121 if let Some(ev) = push_ctx.get(repo.full_name.as_str()) {
122     let branch = ev.payload["ref"]
123         .as_str()
124         .unwrap_or("")
125         .trim_start_matches("refs/heads/");
126     // Prefer a direct link to the HEAD commit; fall back to the branch tree.
127     let push_url = ev.payload["head"]
128         .as_str()
129         .map(|sha| format!("https://github.com/{}/commit/{sha}", repo.full_name))
130         .unwrap_or_else(|| format!("https://github.com/{}/tree/{branch}", repo.full_name));
131     // Commit messages from the event payload (present for normal pushes).
132     let from_payload: Vec<String> = ev.payload["commits"]
133         .as_array()
134         .into_iter()
135         .flatten()
136         .filter_map(|c| c["message"].as_str())
137         .map(|m| m.lines().next().unwrap_or(m).to_string())
138         .take(2)
139         .collect();
140     // Fall back to API-fetched message via HEAD SHA (force push / rebase gave empty payload).
141     let commits = if from_payload.is_empty() {
142         ev.payload["head"]
143             .as_str()
144             .and_then(|sha| commit_msgs.get(sha))
145             .map(|msg| vec![msg.clone()])
146             .unwrap_or_default()
147     } else {
148         from_payload
149     };
150     if commits.is_empty() {
151         let date = ev.created_at.get(..10).unwrap_or(&ev.created_at);
152         builder.write_line(&[Span {
153             text: format!(" \u{2192} pushed to {branch} on {date}"),
154             font_id: italic.clone(),
155             size: Pt(7.5),

```

```

156         color: dark_gray.clone(),
157     });
158     builder.add_link(builder.line_height(), Actions::Uri(push_url));
159 } else {
160     builder.write_line(&[Span {
161         text: format!("\u{2192} pushed to {branch}:"),
162         font_id: italic.clone(),
163         size: Pt(7.5),
164         color: dark_gray.clone(),
165     }]);
166     builder.add_link(builder.line_height(), Actions::Uri(push_url.clone()));
167     commits.iter().for_each(|msg| {
168         builder.write_line(&[Span {
169             text: format!("\u{2192} {msg}"),
170             font_id: italic.clone(),
171             size: Pt(7.5),
172             color: gray.clone(),
173         }]);
174         builder.add_link(builder.line_height(), Actions::Uri(push_url.clone()));
175     });
176 }
177 } else if let Some(ev) = activity_ctx.get(repo.full_name.as_str()) {
178     let date = ev.created_at.get(..10).unwrap_or(&ev.created_at);
179     builder.write_line(&[Span {
180         text: format!("\u{2192} {} on {date}", brief_activity(ev)),
181         font_id: italic.clone(),
182         size: Pt(7.5),
183         color: dark_gray.clone(),
184     }]);
185     builder.add_link(builder.line_height(), Actions::Uri(repo.html_url.clone()));
186 }
187
188     builder.vertical_space(4.0);
189 });
190
191     builder.vertical_space(12.0);
192 }
193
194 /// One-line description of a non-push GitHub event for display in the repo context row.
195 fn brief_activity(event: &GitHubEvent) -> String {
196     let p = &event.payload;
197     match event.kind.as_str() {
198         "PullRequestEvent" => {
199             let action = p["action"].as_str().unwrap_or("updated");
200             let n = p["pull_request"]["number"].as_u64().unwrap_or(0);
201             let merged =
202                 action == "closed" && p["pull_request"]["merged"].as_bool().unwrap_or(false);
203             if merged {
204                 format!("merged PR #{n}")
205             } else {
206                 format!("{action} PR #{n}")
207             }
208         }
209         "IssuesEvent" => {
210             let action = p["action"].as_str().unwrap_or("updated");
211             let n = p["issue"]["number"].as_u64().unwrap_or(0);
212             format!("{action} issue #{n}")
213         }
214         "IssueCommentEvent" => {
215             let n = p["issue"]["number"].as_u64().unwrap_or(0);
216             format!("commented on issue #{n}")
217         }
218         "PullRequestReviewEvent" | "PullRequestReviewCommentEvent" => {
219             let n = p["pull_request"]["number"].as_u64().unwrap_or(0);
220             format!("reviewed PR #{n}")
221         }
222         "WatchEvent" => "starred".to_string(),
223         "ForkEvent" => "forked".to_string(),
224         "ReleaseEvent" => {
225             let tag = p["release"]["tag_name"].as_str().unwrap_or("");
226             format!("released {tag}")
227         }
228         "CreateEvent" => {
229             let ref_type = p["ref_type"].as_str().unwrap_or("ref");
230             let ref_name = p["ref"].as_str().unwrap_or("");
231             if ref_name.is_empty() {
232                 format!("created {ref_type}")
233             } else {

```

```
234         format!("created {ref_type} '{ref_name}'")
235     }
236 }
237 other => other.replace("Event", "").to_lowercase(),
238 }
239 }
240
241 #[cfg(test)]
242 mod tests {
243     use super::*;
244     use crate::pdf;
245     use crate::types::Config;
246
247     fn test_repo(name: &str, stars: u64) -> GitHubRepo {
248         GitHubRepo {
249             name: name.to_string(),
250             full_name: format!("alice/{name}"),
251             html_url: format!("https://github.com/alice/{name}"),
252             description: Some(format!("{name} - a great project")),
253             language: Some("Rust".to_string()),
254             stargazers_count: stars,
255             forks_count: 10,
256             open_issues_count: 3,
257             size: 2048,
258             pushed_at: Some("2024-03-01T00:00:00Z".to_string()),
259             updated_at: Some("2024-03-02T00:00:00Z".to_string()),
260             created_at: Some("2020-06-15T00:00:00Z".to_string()),
261             fork: false,
262         }
263     }
264
265     fn test_push_event(repo: &str, branch: &str, msgs: &[&str]) -> GitHubEvent {
266         use crate::github::EventRepo;
267         let commits: Vec<serde_json::Value> = msgs
268             .iter()
269             .map(|m| serde_json::json!({ "message": m }))
270             .collect();
271         GitHubEvent {
272             kind: "PushEvent".to_string(),
273             repo: EventRepo {
274                 name: repo.to_string(),
275             },
276             payload: serde_json::json!({
277                 "ref": format!("refs/heads/{branch}"),
278                 "commits": commits,
279                 "size": msgs.len()
280             }),
281             created_at: "2024-03-01T09:00:00Z".to_string(),
282         }
283     }
284
285     #[test]
286     fn render_repos_does_not_panic() {
287         let mut doc = printpdf::PdfDocument::new("test");
288         let fonts = pdf::fonts::load_fonts(&mut doc).unwrap();
289         let config = Config::test_default();
290         let mut builder = pdf::create_builder(&config, fonts);
291         super::render(
292             &mut builder,
293             "Top Starred Repositories",
294             &[test_repo("gitprint", 500), test_repo("another", 200)],
295             &[],
296             &std::collections::HashMap::new(),
297         );
298         assert!(!builder.finish().is_empty());
299     }
300
301     #[test]
302     fn render_repos_empty_is_noop() {
303         let mut doc = printpdf::PdfDocument::new("test");
304         let fonts = pdf::fonts::load_fonts(&mut doc).unwrap();
305         let config = Config::test_default();
306         let mut builder = pdf::create_builder(&config, fonts);
307         let page_before = builder.current_page();
308         super::render(
309             &mut builder,
310             "Top Repos",
311             &[],
```

```
312         &[],
313         &std::collections::HashMap::new(),
314     );
315     assert_eq!(builder.current_page(), page_before);
316 }
317
318 #[test]
319 fn render_fork_repo_shows_tag() {
320     let mut doc = printpdf::PdfDocument::new("test");
321     let fonts = pdf::fonts::load_fonts(&mut doc).unwrap();
322     let config = Config::test_default();
323     let mut builder = pdf::create_builder(&config, fonts);
324     let mut repo = test_repo("forked", 5);
325     repo.fork = true;
326     super::render(
327         &mut builder,
328         "Forks",
329         &[repo],
330         &[],
331         &std::collections::HashMap::new(),
332     );
333     assert!(!builder.finish().is_empty());
334 }
335
336 fn test_issue_event(repo: &str, number: u64) -> GitHubEvent {
337     use crate::github::EventRepo;
338     GitHubEvent {
339         kind: "IssuesEvent".to_string(),
340         repo: EventRepo {
341             name: repo.to_string(),
342         },
343         payload: serde_json::json!({ "action": "opened", "issue": { "number": number } }),
344         created_at: "2024-03-02T10:00:00Z".to_string(),
345     }
346 }
347
348 #[test]
349 fn render_repos_with_activity_event_context() {
350     let mut doc = printpdf::PdfDocument::new("test");
351     let fonts = pdf::fonts::load_fonts(&mut doc).unwrap();
352     let config = Config::test_default();
353     let mut builder = pdf::create_builder(&config, fonts);
354     let events = [test_issue_event("alice/gitprint", 42)];
355     super::render(
356         &mut builder,
357         "Repos You Were Active In",
358         &[test_repo("gitprint", 100)],
359         &events,
360         &std::collections::HashMap::new(),
361     );
362     assert!(!builder.finish().is_empty());
363 }
364
365 #[test]
366 fn render_repos_with_push_event_context() {
367     let mut doc = printpdf::PdfDocument::new("test");
368     let fonts = pdf::fonts::load_fonts(&mut doc).unwrap();
369     let config = Config::test_default();
370     let mut builder = pdf::create_builder(&config, fonts);
371     let events = [test_push_event(
372         "alice/gitprint",
373         "main",
374         &["fix: typo", "feat: add feature"],
375     )];
376     super::render(
377         &mut builder,
378         "Recently Pushed",
379         &[test_repo("gitprint", 100)],
380         &events,
381         &std::collections::HashMap::new(),
382     );
383     assert!(!builder.finish().is_empty());
384 }
385
386 #[test]
387 fn render_repos_push_event_no_commits_shows_branch() {
388     let mut doc = printpdf::PdfDocument::new("test");
389     let fonts = pdf::fonts::load_fonts(&mut doc).unwrap();
```

```
390     let config = Config::test_default();
391     let mut builder = pdf::create_builder(&config, fonts);
392     // Push with empty commits array - falls through to "pushed to {branch} on {date}" path.
393     let events = [test_push_event("alice/gitprint", "main", &[])];
394     super::render(
395         &mut builder,
396         "Pushed",
397         &[test_repo("gitprint", 100)],
398         &events,
399         &std::collections::HashMap::new(),
400     );
401     assert(!builder.finish().is_empty());
402 }
403
404 fn make_event(repo: &str, kind: &str, payload: serde_json::Value) -> GitHubEvent {
405     use crate::github::EventRepo;
406     GitHubEvent {
407         kind: kind.to_string(),
408         repo: EventRepo {
409             name: repo.to_string(),
410         },
411         payload,
412         created_at: "2024-03-01T10:00:00Z".to_string(),
413     }
414 }
415
416 #[test]
417 fn brief_activity_pr_open() {
418     let e = make_event(
419         "a/b",
420         "PullRequestEvent",
421         serde_json::json!({ "action": "opened", "pull_request": { "number": 5, "merged": false } })),
422     );
423     assert_eq!(super::brief_activity(&e), "opened PR #5");
424 }
425
426 #[test]
427 fn brief_activity_pr_merged() {
428     let e = make_event(
429         "a/b",
430         "PullRequestEvent",
431         serde_json::json!({ "action": "closed", "pull_request": { "number": 9, "merged": true } })),
432     );
433     assert_eq!(super::brief_activity(&e), "merged PR #9");
434 }
435
436 #[test]
437 fn brief_activity_issues_event() {
438     let e = make_event(
439         "a/b",
440         "IssuesEvent",
441         serde_json::json!({ "action": "opened", "issue": { "number": 3 } })),
442     );
443     assert_eq!(super::brief_activity(&e), "opened issue #3");
444 }
445
446 #[test]
447 fn brief_activity_issue_comment() {
448     let e = make_event(
449         "a/b",
450         "IssueCommentEvent",
451         serde_json::json!({ "issue": { "number": 7 } })),
452     );
453     assert_eq!(super::brief_activity(&e), "commented on issue #7");
454 }
455
456 #[test]
457 fn brief_activity_pr_review() {
458     let e = make_event(
459         "a/b",
460         "PullRequestReviewEvent",
461         serde_json::json!({ "pull_request": { "number": 2 } })),
462     );
463     assert_eq!(super::brief_activity(&e), "reviewed PR #2");
464 }
465
466 #[test]
467 fn brief_activity_watch() {
```

```
468     let e = make_event("a/b", "WatchEvent", serde_json::json!({}));
469     assert_eq!(super::brief_activity(&e), "starred");
470 }
471
472 #[test]
473 fn brief_activity_fork() {
474     let e = make_event("a/b", "ForkEvent", serde_json::json!({}));
475     assert_eq!(super::brief_activity(&e), "forked");
476 }
477
478 #[test]
479 fn brief_activity_release() {
480     let e = make_event(
481         "a/b",
482         "ReleaseEvent",
483         serde_json::json!({ "release": { "tag_name": "v2.0.0" } }),
484     );
485     assert_eq!(super::brief_activity(&e), "released v2.0.0");
486 }
487
488 #[test]
489 fn brief_activity_create_with_ref() {
490     let e = make_event(
491         "a/b",
492         "CreateEvent",
493         serde_json::json!({ "ref_type": "branch", "ref": "hotfix" }),
494     );
495     assert_eq!(super::brief_activity(&e), "created branch 'hotfix'");
496 }
497
498 #[test]
499 fn brief_activity_create_without_ref() {
500     let e = make_event(
501         "a/b",
502         "CreateEvent",
503         serde_json::json!({ "ref_type": "repository", "ref": "" }),
504     );
505     assert_eq!(super::brief_activity(&e), "created repository");
506 }
507
508 #[test]
509 fn brief_activity_unknown_event() {
510     let e = make_event("a/b", "SpookyEvent", serde_json::json!({}));
511     assert!(!super::brief_activity(&e).is_empty());
512 }
513
514 #[test]
515 fn render_repos_no_description() {
516     let mut doc = printpdf::PdfDocument::new("test");
517     let fonts = pdf::fonts::load_fonts(&mut doc).unwrap();
518     let config = Config::test_default();
519     let mut builder = pdf::create_builder(&config, fonts);
520     let mut repo = test_repo("nodesc", 10);
521     repo.description = None;
522     super::render(
523         &mut builder,
524         "Repos",
525         &[repo],
526         &[],
527         &std::collections::HashMap::new(),
528     );
529     assert!(!builder.finish().is_empty());
530 }
531 }
```

## src/preview.rs

```

1  /// Terminal preview mode – renders repository or user data to stdout
2  /// instead of generating a PDF.
3
4  use std::collections::BTreeMap;
5  use std::path::PathBuf;
6  use std::sync::Arc;
7
8  use crate::filter::FileFilter;
9  use crate::git;
10 use crate::github::{CommitDetail, GitHubEvent, GitHubRepo};
11 use crate::types::{Config, UserReportConfig};
12 use crate::user_report::fetch_data;
13 use crate::{format_size, format_utc_now};
14
15 // — ANSI helpers
16
17 struct Ansi {
18     color: bool,
19 }
20
21 impl Ansi {
22     fn new() -> Self {
23         use std::io::IsTerminal;
24         Self {
25             color: std::io::stdout().is_terminal(),
26         }
27     }
28
29     fn bold(&self, s: &str) -> String {
30         if self.color {
31             format!("\x1b[1m{s}\x1b[0m")
32         } else {
33             s.to_string()
34         }
35     }
36
37     fn dim(&self, s: &str) -> String {
38         if self.color {
39             format!("\x1b[2m{s}\x1b[0m")
40         } else {
41             s.to_string()
42         }
43     }
44
45     fn cyan(&self, s: &str) -> String {
46         if self.color {
47             format!("\x1b[1;36m{s}\x1b[0m")
48         } else {
49             s.to_string()
50         }
51     }
52
53     fn yellow(&self, s: &str) -> String {
54         if self.color {
55             format!("\x1b[33m{s}\x1b[0m")
56         } else {
57             s.to_string()
58         }
59     }
60
61     fn green(&self, s: &str) -> String {
62         if self.color {
63             format!("\x1b[32m{s}\x1b[0m")
64         } else {
65             s.to_string()
66         }
67     }
68
69     fn red(&self, s: &str) -> String {
70         if self.color {
71             format!("\x1b[31m{s}\x1b[0m")
72         } else {
73             s.to_string()
74         }
75     }
76
77     fn magenta(&self, s: &str) -> String {

```

```

78     if self.color {
79         format!("\x1b[35m{s}\x1b[0m")
80     } else {
81         s.to_string()
82     }
83 }
84 }
85
86 fn divider(a: &Ansi) -> String {
87     a.dim("&-".repeat(64))
88 }
89
90 fn section_header(a: &Ansi, title: &str) {
91     println!();
92     println!("{}", divider(a));
93     println!();
94     println!(" {}", a.cyan(title));
95     println!();
96 }
97
98 fn box_header(a: &Ansi, title: &str) {
99     let inner = format!(" {} ", title);
100    let width = 64usize.max(inner.len() + 4);
101    let bar = "&-".repeat(width - 2);
102    println!("{}", a.dim(&format!("[{}]", bar)));
103    println!("{}", a.bold(&format!("[{}inner:<pad$|", pad = width - 2)));
104    println!("{}", a.dim(&format!("[{}]", bar)));
105 }
106
107 fn kv(a: &Ansi, key: &str, value: &str) {
108     if value.is_empty() {
109         return;
110     }
111     println!(" {} {}", a.cyan(&format!("[key:<12]", key)), value);
112 }
113
114 fn format_number(n: usize) -> String {
115     let s = n.to_string();
116     let mut result = String::new();
117     s.chars().rev().enumerate().for_each(|(i, c)| {
118         if i > 0 && i % 3 == 0 {
119             result.push(',');
120         }
121         result.push(c);
122     });
123     result.chars().rev().collect()
124 }
125
126 fn fmt_u64(n: u64) -> String {
127     format_number(n as usize)
128 }
129
130 // — Repository preview —————
131
132 /// Previews a repository or file in the terminal.
133 pub async fn repo(config: &Config) -> anyhow::Result<> {
134     let a = Ansi::new();
135     let info = git::verify_repo(&config.repo_path).await?;
136
137     // — Single-file mode —————
138     if let Some(ref single_file) = info.single_file {
139         let (content_res, last_modified) = tokio::join!(
140             git::read_file_content(&info.root, single_file, config),
141             git::file_last_modified(&info.root, single_file, config, info.is_git),
142         );
143         let content = content_res?;
144         let line_count = content.lines().count();
145         let size_str = format_size(content.len() as u64);
146
147         box_header(&a, &single_file.display().to_string());
148         println!();
149         kv(&a, "LINES", &format_number(line_count));
150         kv(&a, "SIZE", &size_str);
151         kv(&a, "MODIFIED", &last_modified);
152         println!();
153         println!("{}", divider(&a));
154         println!();
155         content.lines().enumerate().take(200).for_each(|(i, line)| {

```

```

156     println!("{}", a.dim(&format!("{:4}", i + 1)));
157 }
158 if line_count > 200 {
159     println!(
160         "{}",
161         a.dim(&format!(
162             " ... {} more lines",
163             format_number(line_count - 200)
164         ))
165     );
166 }
167 println!();
168 return Ok(());
169 }
170
171 // — Multi-file / repository mode —————
172 let repo_path = info.root.clone();
173 let is_git = info.is_git;
174 let scope = info.scope.clone();
175 let is_remote = config.remote_url.is_some();
176 let generated_at = format_utc_now();
177 let repo_path2 = repo_path.clone();
178 let config2 = config.clone();
179 let repo_path3 = repo_path.clone();
180
181 let (metadata_res, all_paths_res, date_map_res, fs_owner_group, git_repo_size, fs_size) = tokio::join!(
182     git::get_metadata(&repo_path, config, is_git, scope.as_deref()),
183     git::list_tracked_files(&repo_path, config, is_git, scope.as_deref()),
184     git::file_last_modified_dates(&repo_path, config, is_git, scope.as_deref()),
185     async {
186         if is_remote {
187             (None, None)
188         } else {
189             git::fs_owner_group(&config.repo_path).await
190         }
191     },
192     async {
193         if is_git {
194             git::git_tracked_size(&repo_path2, &config2).await
195         } else {
196             String::new()
197         }
198     },
199     async {
200         if is_remote {
201             String::new()
202         } else {
203             git::fs_dir_size(&repo_path3).await
204         }
205     },
206 );
207
208 let mut metadata = metadata_res?;
209 if let Some(ref url) = config.remote_url {
210     metadata.name = git::repo_name_from_url(url);
211 }
212 metadata.fs_owner = fs_owner_group.0;
213 metadata.fs_group = fs_owner_group.1;
214 metadata.generated_at = generated_at;
215 metadata.repo_size = git_repo_size;
216 metadata.fs_size = fs_size;
217 if !is_remote {
218     metadata.repo_absolute_path = Some(repo_path.clone());
219 }
220
221 let date_map = Arc::new(date_map_res?);
222 let file_filter = FileFilter::new(&config.include_patterns, &config.exclude_patterns)?;
223 let mut paths: Vec<PathBuf> = file_filter.filter_paths(all_paths_res?).collect();
224 paths.sort_unstable();
225
226 // Read file contents in parallel to get LOC + size info.
227 let mut read_set: tokio::task::JoinSet<Option<(PathBuf, usize, String, String)>> =
228     tokio::task::JoinSet::new();
229 paths.iter().for_each(|path| {
230     let p = path.clone();
231     let r = repo_path.clone();
232     let c = config.clone();
233     let dates = Arc::clone(&date_map);

```

```

234 read_set.spawn(async move {
235     let content = git::read_file_content(&r, &p, &c).await.ok()?;
236     if crate::filter::is_binary(content.as_bytes()) || crate::filter::is_minified(&content)
237     {
238         return None;
239     }
240     let line_count = content.lines().count();
241     let size_str = format_size(content.len() as u64);
242     let last_modified = dates.get(&p).cloned().unwrap_or_default();
243     Some((p, line_count, size_str, last_modified))
244 });
245 });
246
247 let mut files: Vec<PathBuf, usize, String, String> =
248     read_set.join_all().await.into_iter().flatten().collect();
249 files.sort_unstable_by(|a, b| a.0.cmp(&b.0));
250
251 metadata.file_count = files.len();
252 metadata.total_lines = files.iter().map(|(_, lc, _, _)| lc).sum();
253
254 let effective_remote = config
255     .remote_url
256     .as_deref()
257     .or(metadata.detected_remote_url.as_deref());
258
259 // — Header —————
260 box_header(&a, &metadata.name);
261 println!();
262
263 let commit_first_line = metadata.commit_message.lines().next().unwrap_or("");
264 let commit_line = format!(
265     "{} · {} · {}",
266     a.yellow(&metadata.commit_hash_short),
267     metadata
268         .commit_date
269         .get(..10)
270         .unwrap_or(&metadata.commit_date),
271     commit_first_line,
272 );
273 kv(&a, "BRANCH", &metadata.branch);
274 kv(&a, "COMMIT", &commit_line);
275 let author = if metadata.commit_author_email.is_empty() {
276     metadata.commit_author.clone()
277 } else {
278     format!(
279         "{} <{}>",
280         metadata.commit_author, metadata.commit_author_email
281     )
282 };
283 kv(&a, "AUTHOR", &author);
284 if let Some(url) = effective_remote {
285     kv(&a, "REMOTE", url);
286 }
287 if let Some(path) = &metadata.repo_absolute_path {
288     kv(&a, "PATH", &path.display().to_string());
289 }
290 let size_info = match (metadata.repo_size.is_empty(), metadata.fs_size.is_empty()) {
291     (false, false) => {
292         format!(
293             "{} (git tracked) · {} (disk)",
294             metadata.repo_size, metadata.fs_size
295         )
296     }
297     (false, true) => metadata.repo_size.clone(),
298     (true, false) => metadata.fs_size.clone(),
299     _ => String::new(),
300 };
301 kv(&a, "SIZE", &size_info);
302 if let (Some(owner), Some(group)) = (&metadata.fs_owner, &metadata.fs_group) {
303     kv(&a, "OWNER", &format!("{owner}:{group}"));
304 }
305 kv(&a, "GENERATED", &metadata.generated_at);
306 println!();
307 println!(
308     " {} {} {} {}",
309     a.cyan("FILES"),
310     a.bold(&format_number(metadata.file_count)),
311     a.cyan("LINES"),

```

```

312     a.bold(&format_number(metadata.total_lines)),
313 );
314
315 // — Directory tree —————
316 if config.file_tree {
317     section_header(&a, "DIRECTORY TREE");
318     let tree_paths: Vec<PathBuf> = files.iter().map(|(p, _, _, _)| p.clone()).collect();
319     print_tree(&a, &tree_paths, &metadata.name);
320 }
321
322 // — File list —————
323 section_header(&a, &format!("FILES  ({})", format_number(files.len())));
324
325 let max_path = files
326     .iter()
327     .map(|(p, _, _, _)| p.display().to_string().len())
328     .max()
329     .unwrap_or(4)
330     .min(60);
331 let max_loc = files
332     .iter()
333     .map(|(_, lc, _, _)| format_number(*lc).len())
334     .max()
335     .unwrap_or(3)
336     .max(3);
337 let max_size = files
338     .iter()
339     .map(|(_, _, s, _)| s.len())
340     .max()
341     .unwrap_or(4)
342     .max(4);
343
344 println!(
345     "  {} {:<path_w$} {:>loc_w$} {:<size_w$} {}",
346     a.dim("  "),
347     a.dim("PATH"),
348     a.dim("LOC"),
349     a.dim("SIZE"),
350     a.dim("MODIFIED"),
351     path_w = max_path,
352     loc_w = max_loc,
353     size_w = max_size,
354 );
355 println!(
356     "  {}",
357     a.dim(&"-".repeat(max_path + max_loc + max_size + 26))
358 );
359
360 files
361     .iter()
362     .enumerate()
363     .for_each(|(i, (path, line_count, size_str, last_modified))| {
364         println!(
365             "  {} {:<path_w$} {:>loc_w$} {:<size_w$} {}",
366             a.dim(&format!("{:4}.", i + 1)),
367             path.display(),
368             a.bold(&format_number(*line_count)),
369             size_str,
370             a.dim(last_modified),
371             path_w = max_path,
372             loc_w = max_loc,
373             size_w = max_size,
374         );
375     });
376
377 println!();
378 Ok(())
379 }
380
381 // — User report preview —————
382
383 /// Previews a GitHub user report in the terminal.
384 pub async fn user(config: &UserReportConfig) -> anyhow::Result<> {
385     let a = Ansi::new();
386     eprintln!("Fetching GitHub data for @{}...", config.username);
387     let data = fetch_data(config).await?;
388
389     // — Header —————

```



```

468
469 // — Commits —————
470 if !data.commit_details.is_empty() {
471     let sha_to_branch: std::collections::HashMap<&str, &str> = data
472         .events
473         .iter()
474         .filter(|e| e.kind == "PushEvent")
475         .filter_map(|e| {
476             let sha = e.payload["head"].as_str()?;
477             let branch = e.payload["ref"].as_str()?.trim_start_matches("refs/heads/");
478             Some((sha, branch))
479         })
480         .collect();
481
482     section_header(&a, "RECENT COMMITS");
483     data.commit_details.iter().for_each(|(repo, detail)| {
484         let branch = sha_to_branch
485             .get(detail.sha.as_str())
486             .copied()
487             .unwrap_or("main");
488         print_commit(&a, repo, detail, branch);
489     });
490 }
491
492 println!();
493 Ok(())
494 }
495
496 // — Event printer —————
497
498 fn print_event(
499     a: &Ansi,
500     event: &GitHubEvent,
501     commit_msgs: &std::collections::HashMap<String, String>,
502 ) {
503     let date = event.created_at.get(..10).unwrap_or(&event.created_at);
504     let repo = &event.repo.name;
505
506     match event.kind.as_str() {
507         "PushEvent" => {
508             let branch = event.payload["ref"]
509                 .as_str()
510                 .unwrap_or("")
511                 .trim_start_matches("refs/heads/");
512             let head_sha = event.payload["head"].as_str().unwrap_or("");
513             let msg = head_sha
514                 .get(..7)
515                 .and_then(|short| commit_msgs.get(short).or_else(|| commit_msgs.get(head_sha)))
516                 .map(|s| s.as_str())
517                 .unwrap_or("");
518             let commit_count = event.payload["commits"]
519                 .as_array()
520                 .map(|c| c.len())
521                 .unwrap_or(0);
522             print!(
523                 " {} {} {} {} ",
524                 a.dim(date),
525                 a.bold("push "),
526                 a.cyan(repo),
527                 a.dim(&format!("· {branch}")),
528             );
529             if commit_count > 0 {
530                 print!(" {} ", a.dim(&format!("[{commit_count} commit(s)]")));
531             }
532             println!();
533             if !msg.is_empty() {
534                 println!(
535                     "         {} {} ",
536                     a.dim(&head_sha[..7.min(head_sha.len())]),
537                     msg
538                 );
539             }
540         }
541         "PullRequestEvent" => {
542             let action = event.payload["action"].as_str().unwrap_or("?");
543             let title = event.payload["pull_request"]["title"]
544                 .as_str()
545                 .unwrap_or("");

```

```
546     let number = event.payload["number"].as_u64().unwrap_or(0);
547     println!(
548         " {} {} {} {}",
549         a.dim(date),
550         a.magenta("pr  "),
551         a.cyan(repo),
552         a.dim(&format!("#{number} {action}")),
553     );
554     if !title.is_empty() {
555         println!("          {title}");
556     }
557 }
558 "IssuesEvent" => {
559     let action = event.payload["action"].as_str().unwrap_or("?");
560     let title = event.payload["issue"]["title"].as_str().unwrap_or("");
561     let number = event.payload["issue"]["number"].as_u64().unwrap_or(0);
562     println!(
563         " {} {} {} {}",
564         a.dim(date),
565         a.yellow("issue  "),
566         a.cyan(repo),
567         a.dim(&format!("#{number} {action}")),
568     );
569     if !title.is_empty() {
570         println!("          {title}");
571     }
572 }
573 "IssueCommentEvent" => {
574     let number = event.payload["issue"]["number"].as_u64().unwrap_or(0);
575     println!(
576         " {} {} {} {}",
577         a.dim(date),
578         a.dim("comment"),
579         a.cyan(repo),
580         a.dim(&format!("issue #{number}")),
581     );
582 }
583 "CreateEvent" => {
584     let ref_type = event.payload["ref_type"].as_str().unwrap_or("?");
585     let ref_name = event.payload["ref"].as_str().unwrap_or("");
586     println!(
587         " {} {} {} {}",
588         a.dim(date),
589         a.green("create  "),
590         a.cyan(repo),
591         a.dim(&format!("{ref_type} {ref_name}")),
592     );
593 }
594 "DeleteEvent" => {
595     let ref_type = event.payload["ref_type"].as_str().unwrap_or("?");
596     let ref_name = event.payload["ref"].as_str().unwrap_or("");
597     println!(
598         " {} {} {} {}",
599         a.dim(date),
600         a.red("delete  "),
601         a.cyan(repo),
602         a.dim(&format!("{ref_type} {ref_name}")),
603     );
604 }
605 "ForkEvent" => {
606     let forkee = event.payload["forkee"]["full_name"].as_str().unwrap_or("");
607     println!(
608         " {} {} {} {}",
609         a.dim(date),
610         a.dim("fork  "),
611         a.cyan(repo),
612         a.dim(&format!("> {forkee}")),
613     );
614 }
615 "WatchEvent" => {
616     println!(" {} {} {} ", a.dim(date), a.dim("star  "), a.cyan(repo),);
617 }
618 "ReleaseEvent" => {
619     let tag = event.payload["release"]["tag_name"].as_str().unwrap_or("");
620     let name = event.payload["release"]["name"].as_str().unwrap_or("");
621     println!(
622         " {} {} {} {}",
623         a.dim(date),
```

```
624     a.green("release"),
625     a.cyan(repo),
626     a.dim(if name.is_empty() { tag } else { name })),
627 );
628 }
629 "PullRequestReviewEvent" => {
630     let state = event.payload["review"]["state"].as_str().unwrap_or("?");
631     let number = event.payload["pull_request"]["number"]
632         .as_u64()
633         .unwrap_or(0);
634     println!(
635         " {} {} {} {}",
636         a.dim(date),
637         a.dim("review "),
638         a.cyan(repo),
639         a.dim(&format!("pr #{number} {state}")),
640     );
641 }
642 other => {
643     let label = other.trim_end_matches("Event");
644     println!(
645         " {} {} {}",
646         a.dim(date),
647         a.dim(&format!("{label:<7}")),
648         a.cyan(repo),
649     );
650 }
651 }
652 }
653
654 // — Repo printer —————
655
656 fn print_repo(a: &Ansi, repo: &GitHubRepo) {
657     let lang = repo.language.as_deref().unwrap_or("-");
658     let desc = repo.description.as_deref().unwrap_or("");
659     let pushed = repo
660         .pushed_at
661         .as_deref()
662         .and_then(|d| d.get(..10))
663         .unwrap_or("");
664
665     println!(
666         " {:<42} {} {:>7} {} {}",
667         a.bold(&repo.full_name),
668         a.dim(&format!("{lang:<12}")),
669         a.yellow(&format!("{:0} {}", fmt_u64(repo.stargazers_count))),
670         a.dim(&format!("{:0} {}", fmt_u64(repo.forks_count))),
671         a.dim(pushed),
672     );
673     if !desc.is_empty() {
674         println!(" {} ", a.dim(desc));
675     }
676 }
677
678 // — Commit printer —————
679
680 fn print_commit(a: &Ansi, repo: &str, detail: &CommitDetail, branch: &str) {
681     let short_sha = detail.sha.get(..7).unwrap_or(&detail.sha);
682     let date = detail
683         .commit
684         .author
685         .date
686         .get(..10)
687         .unwrap_or(&detail.commit.author.date);
688     let msg = detail.commit.message.lines().next().unwrap_or("");
689     let author = &detail.commit.author.name;
690
691     println!(
692         " {} {} {} {} {} ",
693         a.yellow(short_sha),
694         a.dim(date),
695         a.cyan(repo),
696         a.dim(&format!("{branch}")),
697         a.bold(msg),
698     );
699     println!(" {} {} ", a.dim(" ", a.dim(author)));
700
701     let max_fname = detail
```

```

702     .files
703     .iter()
704     .map(|f| f.filename.len())
705     .max()
706     .unwrap_or(0)
707     .min(50);
708
709 detail.files.iter().for_each(|f| {
710     let status_label = match f.status.as_str() {
711         "added" => a.green("added  "),
712         "removed" | "deleted" => a.red("deleted "),
713         "renamed" => a.magenta("renamed "),
714         _ => a.dim("modified"),
715     };
716     println!(
717         " {} {:<fname_w$} {} {}",
718         a.dim("    "),
719         f.filename,
720         status_label,
721         a.dim(&format!(
722             "+{:<5} -{:<5}",
723             fmt_u64(f.additions),
724             fmt_u64(f.deletions)
725         )),
726         fname_w = max_fname,
727     );
728 });
729 println!();
730 }
731
732 // — Tree renderer —————
733
734 struct TreeNode {
735     children: BTreeMap<String, TreeNode>,
736     is_file: bool,
737 }
738
739 fn build_tree(paths: &[PathBuf]) -> TreeNode {
740     let mut root = TreeNode {
741         children: BTreeMap::new(),
742         is_file: false,
743     };
744     for path in paths {
745         let components: Vec<String> = path
746             .components()
747             .map(|c| c.as_os_str().to_string_lossy().into_owned())
748             .collect();
749         let total = components.len();
750         let mut node = &mut root;
751         for (i, name) in components.into_iter().enumerate() {
752             node = node.children.entry(name).or_insert(TreeNode {
753                 children: BTreeMap::new(),
754                 is_file: i == total - 1,
755             });
756         }
757     }
758     root
759 }
760
761 fn print_tree(a: &Ansi, paths: &[PathBuf], root_name: &str) {
762     let tree = build_tree(paths);
763     println!(" {}/", a.bold(root_name));
764     print_tree_children(a, &tree, " ");
765 }
766
767 fn print_tree_children(a: &Ansi, node: &TreeNode, prefix: &str) {
768     // Directories first (sorted), then files (sorted) — BTreeMap is already sorted.
769     let dirs: Vec<(&String, &TreeNode)> =
770         node.children.iter().filter(|(_, n)| !n.is_file).collect();
771     let files: Vec<(&String, &TreeNode)> =
772         node.children.iter().filter(|(_, n)| n.is_file).collect();
773
774     let children: Vec<(&String, &TreeNode)> = dirs.into_iter().chain(files).collect();
775     let last = children.len().saturating_sub(1);
776
777     children.iter().enumerate().for_each(|(i, (name, child))| {
778         let (connector, continuation) = if i == last {
779             ("└─ ", " ")

```

```
780     } else {
781         ("├─ ", "│  ")
782     };
783     if child.is_file {
784         println!("{prefix}{connector}{name}");
785     } else {
786         println!("{prefix}{connector}{}/", a.bold(name));
787         print_tree_children(a, child, &format!("{prefix}{continuation}"));
788     }
789 });
790 }
791
792 #[cfg(test)]
793 mod tests {
794     use super::*;
795     use tempfile::TempDir;
796
797     // — format_number —————
798
799     #[test]
800     fn format_number_zero() {
801         assert_eq!(format_number(0), "0");
802     }
803
804     #[test]
805     fn format_number_small() {
806         assert_eq!(format_number(42), "42");
807         assert_eq!(format_number(999), "999");
808     }
809
810     #[test]
811     fn format_number_thousands() {
812         assert_eq!(format_number(1_000), "1,000");
813         assert_eq!(format_number(12_345), "12,345");
814     }
815
816     #[test]
817     fn format_number_millions() {
818         assert_eq!(format_number(1_000_000), "1,000,000");
819         assert_eq!(format_number(1_234_567), "1,234,567");
820     }
821
822     #[test]
823     fn fmt_u64_rounds_trip() {
824         assert_eq!(fmt_u64(0), "0");
825         assert_eq!(fmt_u64(1_000), "1,000");
826         assert_eq!(fmt_u64(1_000_000), "1,000,000");
827     }
828
829     // — build_tree —————
830
831     #[test]
832     fn build_tree_empty() {
833         let tree = build_tree(&[]);
834         assert!(tree.children.is_empty());
835         assert!(!tree.is_file);
836     }
837
838     #[test]
839     fn build_tree_single_top_level_file() {
840         let tree = build_tree(&[PathBuf::from("main.rs")]);
841         assert_eq!(tree.children.len(), 1);
842         let node = &tree.children["main.rs"];
843         assert!(node.is_file);
844         assert!(node.children.is_empty());
845     }
846
847     #[test]
848     fn build_tree_nested_file() {
849         let tree = build_tree(&[PathBuf::from("src/lib.rs")]);
850         assert_eq!(tree.children.len(), 1);
851         let src = &tree.children["src"];
852         assert!(!src.is_file);
853         assert_eq!(src.children.len(), 1);
854         assert!(src.children["lib.rs"].is_file);
855     }
856
857     #[test]
```

```

858 fn build_tree_dirs_and_files_separated() {
859     let paths = vec![
860         PathBuf::from("README.md"),
861         PathBuf::from("src/main.rs"),
862         PathBuf::from("src/lib.rs"),
863     ];
864     let tree = build_tree(&paths);
865     assert_eq!(tree.children.len(), 2);
866     assert!(!tree.children["src"].is_file);
867     assert!(tree.children["README.md"].is_file);
868     assert_eq!(tree.children["src"].children.len(), 2);
869 }
870
871 #[test]
872 fn build_tree_deeply_nested() {
873     let paths = vec![PathBuf::from("a/b/c/deep.rs")];
874     let tree = build_tree(&paths);
875     let a = &tree.children["a"];
876     assert!(!a.is_file);
877     let b = &a.children["b"];
878     assert!(!b.is_file);
879     let c = &b.children["c"];
880     assert!(!c.is_file);
881     assert!(c.children["deep.rs"].is_file);
882 }
883
884 #[test]
885 fn build_tree_sorted_alphabetically() {
886     let paths = vec![
887         PathBuf::from("z.rs"),
888         PathBuf::from("a.rs"),
889         PathBuf::from("m.rs"),
890     ];
891     let tree = build_tree(&paths);
892     let keys: Vec<&String> = tree.children.keys().collect();
893     assert_eq!(keys, vec!["a.rs", "m.rs", "z.rs"]);
894 }
895
896 #[test]
897 fn build_tree_duplicate_paths_idempotent() {
898     let paths = vec![PathBuf::from("src/main.rs"), PathBuf::from("src/main.rs")];
899     let tree = build_tree(&paths);
900     assert_eq!(tree.children["src"].children.len(), 1);
901 }
902
903 // — preview::repo —————
904 // Verify the full preview pipeline returns Ok on the same inputs as the PDF
905 // pipeline. Stdout output is not captured - Ok(()) is the meaningful contract.
906
907 async fn git(dir: &std::path::Path, args: &[&str]) -> anyhow::Result<> {
908     let p = dir
909         .to_str()
910         .ok_or_else(|| anyhow::anyhow!("non-utf8 path"));
911     tokio::process::Command::new("git")
912         .args(["-C", p])
913         .args(args)
914         .output()
915         .await?;
916     Ok(())
917 }
918
919 fn base_config(repo_path: std::path::PathBuf) -> crate::types::Config {
920     crate::types::Config {
921         repo_path,
922         output_path: std::path::PathBuf::from("/tmp/unused.pdf"),
923         include_patterns: vec![],
924         exclude_patterns: vec![],
925         theme: "InspiredGitHub".to_string(),
926         font_size: 8.0,
927         no_line_numbers: false,
928         toc: true,
929         file_tree: true,
930         branch: None,
931         commit: None,
932         paper_size: crate::types::PaperSize::A4,
933         landscape: false,
934         remote_url: None,
935     }

```

```
936     }
937
938     #[tokio::test]
939     async fn preview_repo_git_repo() -> anyhow::Result<()> {
940         let dir = TempDir::new()?;
941         git(dir.path(), &["init", "-b", "main"]).await?;
942         git(dir.path(), &["config", "user.email", "t@t.com"]).await?;
943         git(dir.path(), &["config", "user.name", "T"]).await?;
944         tokio::fs::write(dir.path().join("main.rs"), "fn main() {}\\n").await?;
945         git(dir.path(), &["add", "."]).await?;
946         git(dir.path(), &["commit", "-m", "init"]).await?;
947
948         crate::preview::repo(&base_config(dir.path().to_path_buf())).await
949     }
950
951     #[tokio::test]
952     async fn preview_repo_plain_directory() -> anyhow::Result<()> {
953         let dir = TempDir::new()?;
954         tokio::fs::write(dir.path().join("hello.rs"), "fn main() {}\\n").await?;
955
956         crate::preview::repo(&base_config(dir.path().to_path_buf())).await
957     }
958
959     #[tokio::test]
960     async fn preview_repo_empty_directory() -> anyhow::Result<()> {
961         let dir = TempDir::new()?;
962         crate::preview::repo(&base_config(dir.path().to_path_buf())).await
963     }
964
965     #[tokio::test]
966     async fn preview_repo_single_file_plain() -> anyhow::Result<()> {
967         let dir = TempDir::new()?;
968         let file = dir.path().join("main.rs");
969         tokio::fs::write(&file, "fn main() { println!(\\\"hi\\\"); }\\n").await?;
970
971         crate::preview::repo(&crate::types::Config {
972             repo_path: file,
973             ..base_config(dir.path().to_path_buf())
974         })
975         .await
976     }
977
978     #[tokio::test]
979     async fn preview_repo_single_file_in_git_repo() -> anyhow::Result<()> {
980         let dir = TempDir::new()?;
981         git(dir.path(), &["init", "-b", "main"]).await?;
982         git(dir.path(), &["config", "user.email", "t@t.com"]).await?;
983         git(dir.path(), &["config", "user.name", "T"]).await?;
984         tokio::fs::write(dir.path().join("lib.rs"), "pub fn f() {}\\n").await?;
985         git(dir.path(), &["add", "."]).await?;
986         git(dir.path(), &["commit", "-m", "init"]).await?;
987
988         crate::preview::repo(&crate::types::Config {
989             repo_path: dir.path().join("lib.rs"),
990             ..base_config(dir.path().to_path_buf())
991         })
992         .await
993     }
994
995     #[tokio::test]
996     async fn preview_repo_no_file_tree() -> anyhow::Result<()> {
997         let dir = TempDir::new()?;
998         tokio::fs::write(dir.path().join("a.rs"), "fn a() {}\\n").await?;
999
1000         crate::preview::repo(&crate::types::Config {
1001             file_tree: false,
1002             ..base_config(dir.path().to_path_buf())
1003         })
1004         .await
1005     }
1006
1007     #[tokio::test]
1008     async fn preview_repo_include_filter() -> anyhow::Result<()> {
1009         let dir = TempDir::new()?;
1010         tokio::try_join!(
1011             tokio::fs::write(dir.path().join("main.rs"), "fn main() {}\\n"),
1012             tokio::fs::write(dir.path().join("README.md"), "# hi\\n"),
1013         )?;
```

```
1014
1015     crate::preview::repo(&crate::types::Config {
1016         include_patterns: vec!["*.rs".to_string()],
1017         ..base_config(dir.path().to_path_buf())
1018     })
1019     .await
1020 }
1021
1022 #[tokio::test]
1023 async fn preview_repo_nonexistent_path_errors() {
1024     assert!(
1025         crate::preview::repo(&base_config(std::path::PathBuf::from(
1026             "/nonexistent/preview/path"
1027         )))
1028         .await
1029         .is_err()
1030     );
1031 }
1032 }
```

## src/types.rs

```

1  use std::path::PathBuf;
2
3  /// Activity filter for the user report event feed.
4  #[derive(Debug, Clone, Copy, PartialEq, Eq, clap::ValueEnum)]
5  pub enum ActivityFilter {
6      /// Show all event types (pushes, PRs, issues, stars, etc.)
7      ALL,
8      /// Show only push events (commits to repos)
9      Commits,
10 }
11
12 /// Configuration for a `gitprint user` run.
13 #[allow(missing_docs)]
14 #[derive(Debug, Clone)]
15 pub struct UserReportConfig {
16     pub username: String,
17     pub output_path: PathBuf,
18     pub paper_size: PaperSize,
19     pub landscape: bool,
20     /// Number of most-recently-pushed repos to include (0 = skip section).
21     pub last_repos: usize,
22     /// Number of recent commits with diffs to render (0 = skip diffs).
23     pub last_commits: usize,
24     /// Skip diff rendering entirely.
25     pub no_diffs: bool,
26     /// Font size used for diff/code blocks.
27     pub font_size: f64,
28     /// GitHub personal access token (`GITHUB_TOKEN` env var).
29     pub github_token: Option<String>,
30     /// Earliest date to include events from, in `YYYY-MM-DD` form (`None` = no lower bound).
31     pub since: Option<String>,
32     /// Latest date to include events from, in `YYYY-MM-DD` form (`None` = no upper bound).
33     pub until: Option<String>,
34     /// Which event types to include in the report.
35     pub activity: ActivityFilter,
36     /// Maximum number of events to show in the activity feed.
37     pub events: usize,
38 }
39
40 /// Paper size for PDF output.
41 #[derive(Debug, Clone, Copy, clap::ValueEnum)]
42 pub enum PaperSize {
43     /// ISO A4 (210 × 297 mm).
44     A4,
45     /// US Letter (215.9 × 279.4 mm).
46     Letter,
47     /// US Legal (215.9 × 355.6 mm).
48     Legal,
49 }
50
51 /// Configuration for a gitprint run.
52 #[allow(missing_docs)]
53 #[derive(Debug, Clone)]
54 pub struct Config {
55     pub repo_path: PathBuf,
56     pub output_path: PathBuf,
57     pub include_patterns: Vec<String>,
58     pub exclude_patterns: Vec<String>,
59     pub theme: String,
60     pub font_size: f64,
61     pub no_line_numbers: bool,
62     pub toc: bool,
63     pub file_tree: bool,
64     pub branch: Option<String>,
65     pub commit: Option<String>,
66     pub paper_size: PaperSize,
67     pub landscape: bool,
68     /// Original remote URL when input was a remote repository, used for GitHub links.
69     pub remote_url: Option<String>,
70 }
71
72 impl Config {
73     #[cfg(test)]
74     pub(crate) fn test_default() -> Self {
75         Self {
76             repo_path: PathBuf::from("."),
77             output_path: PathBuf::from("/tmp/gitprint-test.pdf"),

```

```

78         include_patterns: vec![],
79         exclude_patterns: vec![],
80         theme: "InspiredGitHub".to_string(),
81         font_size: 8.0,
82         no_line_numbers: false,
83         toc: true,
84         file_tree: true,
85         branch: None,
86         commit: None,
87         paper_size: PaperSize::A4,
88         landscape: false,
89         remote_url: None,
90     }
91 }
92 }
93
94 /// Metadata extracted from a git repository.
95 #[allow(missing_docs)]
96 #[derive(Debug, Clone)]
97 pub struct RepoMetadata {
98     pub name: String,
99     pub branch: String,
100    pub commit_hash: String,
101    pub commit_hash_short: String,
102    pub commit_date: String,
103    pub commit_message: String,
104    pub commit_author: String,
105    /// Email address of the last committer.
106    pub commit_author_email: String,
107    pub file_count: usize,
108    pub total_lines: usize,
109    /// Filesystem owner of the input path (local paths only).
110    pub fs_owner: Option<String>,
111    /// Filesystem group of the input path (local paths only).
112    pub fs_group: Option<String>,
113    /// UTC timestamp when this PDF was generated.
114    pub generated_at: String,
115    /// Human-readable size of the git-tracked content (e.g. "4.2 MB").
116    /// Computed from `git ls-tree -r -l`; empty for non-git paths.
117    pub repo_size: String,
118    /// Human-readable filesystem disk usage of the input path (e.g. "5.1 MB").
119    /// Computed from `du -sh`; empty for remote repos.
120    pub fs_size: String,
121    /// Remote URL detected from git config (e.g. `git remote get-url origin`).
122    /// Used to generate commit/author links even when `Config::remote_url` is None.
123    pub detected_remote_url: Option<String>,
124    /// Absolute filesystem path to the repo root (local repos only, `None` for remote clones).
125    /// Used to generate `file:///` links on the cover page.
126    pub repo_absolute_path: Option<PathBuf>,
127 }
128
129 /// An RGB color value.
130 #[allow(missing_docs)]
131 #[derive(Debug, Clone, Copy)]
132 pub struct RgbColor {
133     pub r: u8,
134     pub g: u8,
135     pub b: u8,
136 }
137
138 /// A single syntax-highlighted token with styling information.
139 #[allow(missing_docs)]
140 #[derive(Debug, Clone)]
141 pub struct HighlightedToken {
142     pub text: String,
143     pub color: RgbColor,
144     pub bold: bool,
145     pub italic: bool,
146 }
147
148 /// A line of syntax-highlighted tokens.
149 #[allow(missing_docs)]
150 #[derive(Debug, Clone)]
151 pub struct HighlightedLine {
152     pub line_number: usize,
153     pub tokens: Vec<HighlightedToken>,
154 }
155

```

```
156 #[cfg(test)]
157 mod tests {
158     use super::*;
159
160     #[test]
161     fn test_config_test_default() {
162         let config = Config::test_default();
163         assert_eq!(config.repo_path, PathBuf::from("."));
164         assert_eq!(config.theme, "InspiredGitHub");
165         assert_eq!(config.font_size, 8.0);
166         assert!(config.toc);
167         assert!(config.file_tree);
168         assert!(!config.no_line_numbers);
169         assert!(!config.landscape);
170         assert!(config.branch.is_none());
171         assert!(config.commit.is_none());
172     }
173
174     #[test]
175     fn test_repo_metadata_clone() {
176         let meta = RepoMetadata {
177             name: "test".to_string(),
178             branch: "main".to_string(),
179             commit_hash: "abc123".to_string(),
180             commit_hash_short: "abc1234".to_string(),
181             commit_date: "2024-01-01".to_string(),
182             commit_message: "init".to_string(),
183             commit_author: "Alice".to_string(),
184             commit_author_email: "alice@example.com".to_string(),
185             file_count: 10,
186             total_lines: 500,
187             fs_owner: None,
188             fs_group: None,
189             generated_at: "2024-01-15 10:00:00 UTC".to_string(),
190             repo_size: "1.2 MB".to_string(),
191             fs_size: "1.5 MB".to_string(),
192             detected_remote_url: None,
193             repo_absolute_path: None,
194         };
195         let cloned = meta.clone();
196         assert_eq!(cloned.name, "test");
197         assert_eq!(cloned.file_count, 10);
198     }
199
200     #[test]
201     fn test_rgb_color_copy() {
202         let color = RgbColor {
203             r: 255,
204             g: 128,
205             b: 0,
206         };
207         let copied = color;
208         assert_eq!(copied.r, 255);
209         assert_eq!(copied.g, 128);
210         assert_eq!(copied.b, 0);
211         // Original still usable (Copy trait)
212         assert_eq!(color.r, 255);
213     }
214
215     #[test]
216     fn test_highlighted_line_structure() {
217         let line = HighlightedLine {
218             line_number: 42,
219             tokens: vec![
220                 HighlightedToken {
221                     text: "fn".to_string(),
222                     color: RgbColor { r: 0, g: 0, b: 255 },
223                     bold: true,
224                     italic: false,
225                 },
226                 HighlightedToken {
227                     text: " main".to_string(),
228                     color: RgbColor { r: 0, g: 0, b: 0 },
229                     bold: false,
230                     italic: false,
231                 },
232             ],
233         };
234     }
235 }
```

```
234     assert_eq!(line.line_number, 42);
235     assert_eq!(line.tokens.len(), 2);
236     assert!(line.tokens[0].bold);
237     assert!(!line.tokens[1].bold);
238 }
239 }
```

## src/user\_report.rs

```

1  /// User report pipeline: fetch GitHub data in parallel, then render PDF.
2
3  use tokio::task::JoinSet;
4
5  use crate::github::{self, CommitDetail, GitHubEvent, GitHubRepo, GitHubUser};
6  use crate::pdf;
7  use crate::types::{ActivityFilter, UserReportConfig};
8
9  /// Pre-fetched GitHub data consumed by the PDF render phase.
10 ///
11 /// Separating the fetch phase from the render phase keeps the render logic
12 /// testable without any network I/O.
13 pub(crate) struct UserReportData {
14     pub user: GitHubUser,
15     pub total_stars: u64,
16     pub starred_repos: Vec<GitHubRepo>,
17     pub active_repos: Vec<GitHubRepo>,
18     pub pushed_repos: Vec<GitHubRepo>,
19     pub events: Vec<GitHubEvent>,
20     pub commit_msgs: std::collections::HashMap<String, String>,
21     pub commit_details: Vec<(String, CommitDetail)>,
22 }
23
24 /// Fetches all GitHub data for the user report (Phases 1 & 2).
25 ///
26 /// Separated from [run] so that [crate::preview] can reuse the same fetch
27 /// logic without triggering PDF rendering.
28 pub(crate) async fn fetch_data(config: &UserReportConfig) -> anyhow::Result<UserReportData> {
29     let token = config.github_token.as_deref();
30     let username = &config.username;
31
32     // — Phase 1: parallel API fetches —————
33     let (user_res, starred_res, active_res, pushed_res, events_res, search_commits_res) = tokio::join!(
34         github::get_user(username, token),
35         github::get_user_starred_repos(username, 5, token),
36         github::get_user_repos(username, "updated", 5, token),
37         github::get_user_repos(username, "pushed", config.last_repos, token),
38         github::get_user_events(username, 100, token),
39         async {
40             if config.no_diffs || config.last_commits == 0 {
41                 Ok(vec![])
42             } else {
43                 github::search_user_commits(username, config.last_commits, token).await
44             }
45         },
46     );
47
48     let user = user_res?;
49     let starred_repos = starred_res.unwrap_or_default();
50
51     let events = {
52         let raw = coalesce_push_events(events_res.unwrap_or_default());
53         let date_filtered = raw.into_iter().filter(|e| {
54             let date = e.created_at.get(..10).unwrap_or(&e.created_at);
55             config.since.as_deref().is_none_or(|s| date >= s)
56             && config.until.as_deref().is_none_or(|u| date <= u)
57         });
58         match config.activity {
59             ActivityFilter::All => date_filtered.collect:::<Vec<_>>(),
60             ActivityFilter::Commits => date_filtered
61                 .filter(|e| e.kind == "PushEvent")
62                 .collect:::<Vec<_>>(),
63         }
64     };
65
66     let push_event_repos: std::collections::HashSet<String> = events
67         .iter()
68         .filter(|e| e.kind == "PushEvent")
69         .map(|e| e.repo.name.clone())
70         .collect();
71     let other_event_repos: std::collections::HashSet<String> = events
72         .iter()
73         .filter(|e| e.kind != "PushEvent")
74         .map(|e| e.repo.name.clone())
75         .collect();
76
77     let pushed_repos: Vec<_> = pushed_res

```

```

78     .unwrap_or_default()
79     .into_iter()
80     .filter(|r| {
81         !r.fork && (push_event_repos.is_empty() || push_event_repos.contains(&r.full_name))
82     })
83     .collect();
84
85 let pushed_names: std::collections::HashSet<&str> =
86     pushed_repos.iter().map(|r| r.full_name.as_str()).collect();
87 let active_repos: Vec<_> = active_res
88     .unwrap_or_default()
89     .into_iter()
90     .filter(|r| {
91         !r.fork
92         && !pushed_names.contains(r.full_name.as_str())
93         && (other_event_repos.is_empty() || other_event_repos.contains(&r.full_name))
94     })
95     .collect();
96
97 let total_stars: u64 = starred_repos.iter().map(|r| r.stargazers_count).sum();
98
99 // — Phase 2: fetch commit details in parallel —————
100 let search_commits = match search_commits_res {
101     Ok(commits) => commits,
102     Err(e) if e.to_string().contains("rate limit") => return Err(e),
103     Err(_) => vec![],
104 };
105 let commit_msgs: std::collections::HashMap<String, String> = search_commits
106     .iter()
107     .map(|(_, sha, msg)| (sha.clone(), msg.clone()))
108     .collect();
109
110 let commit_details: Vec<(String, CommitDetail)> = if !config.no_diffs && config.last_commits > 0
111 {
112     let shas: Vec<(String, String)> = search_commits
113         .into_iter()
114         .map(|(repo, sha, _)| (repo, sha))
115         .collect();
116     eprintln!("Fetching {} commit diff(s)...", shas.len());
117     let mut set: JoinSet<anyhow::Result<(String, CommitDetail)>> = JoinSet::new();
118     shas.into_iter().for_each(|(repo, sha)| {
119         let tok = token.map(str::to_string);
120         set.spawn(async move {
121             github::get_commit_detail(&repo, &sha, tok.as_deref())
122                 .await
123                 .map(|cd| (repo, cd))
124         });
125     });
126     let mut details: Vec<(String, CommitDetail)> = set
127         .join_all()
128         .await
129         .into_iter()
130         .filter_map(|r| match r {
131             Ok(pair) => Some(Ok(pair)),
132             Err(e) if e.to_string().contains("rate limit") => Some(Err(e)),
133             Err(_) => None,
134         })
135         .collect::<anyhow::Result<Vec<_>>>()?;
136     details.sort_unstable_by(|(_, a), (_, b)| b.commit.author.date.cmp(&a.commit.author.date));
137     details
138 } else {
139     vec![]
140 };
141
142 Ok(UserReportData {
143     user,
144     total_stars,
145     starred_repos,
146     active_repos,
147     pushed_repos,
148     events,
149     commit_msgs,
150     commit_details,
151 })
152 }
153
154 /// Runs the full user report pipeline and writes a PDF to `config.output_path`.
155 pub async fn run(config: &UserReportConfig) -> anyhow::Result<> {

```

```

156 let start = std::time::Instant::now();
157
158 eprintln!("Fetching GitHub data for @{}...", config.username);
159 let data = fetch_data(config).await?;
160
161 eprintln!("Rendering PDF...");
162 let (doc, total_pages) = render_to_doc(config, &data)?;
163 pdf::save_pdf(&doc, &config.output_path).await?;
164
165 let elapsed = elapsed_str(start.elapsed());
166 let pdf_size = tokio::fs::metadata(&config.output_path)
167     .await
168     .map(|m| m.len())
169     .unwrap_or(0);
170 eprintln!(
171     "{} - {} pages, {}, {}",
172     config.output_path.display(),
173     total_pages,
174     format_size(pdf_size),
175     elapsed,
176 );
177 Ok(())
178 }
179
180 /// Render the user report PDF from pre-fetched data.
181 ///
182 /// Returns the assembled `PdfDocument` (ready to save) and the page count.
183 /// No network I/O is performed - all data must be supplied via `data`.
184 pub(crate) fn render_to_doc(
185     config: &UserReportConfig,
186     data: &UserReportData,
187 ) -> anyhow::Result<(printpdf::PdfDocument, usize)> {
188     let mut doc = printpdf::PdfDocument::new(&format!("{}", "GitHub User Report", config.username));
189     let fonts = pdf::fonts::load_fonts(&mut doc)?;
190     let mut builder = pdf::create_user_builder(config, fonts);
191
192     // Cover page
193     pdf::user_cover::render(&mut builder, &data.user, data.total_stars);
194
195     // Activity feed - capped to the requested display limit.
196     let display_events = &data.events[..config.events.min(data.events.len())];
197     pdf::user_activity::render(&mut builder, display_events, &data.commit_msgs);
198
199     // Repository sections - pass events + fetched commit msgs for rich context
200     render_repos_section(
201         &mut builder,
202         "Top Starred Repositories",
203         &data.starred_repos,
204         5,
205         &data.events,
206         &data.commit_msgs,
207     );
208     render_repos_section(
209         &mut builder,
210         "Repos You Were Active In",
211         &data.active_repos,
212         5,
213         &data.events,
214         &data.commit_msgs,
215     );
216     render_repos_section(
217         &mut builder,
218         "Repos User Pushed To",
219         &data.pushed_repos,
220         config.last_repos,
221         &data.events,
222         &data.commit_msgs,
223     );
224
225     // Commit diffs
226     if !data.commit_details.is_empty() {
227         // Build SHA → branch from push events so each diff header can show the branch.
228         let sha_to_branch: std::collections::HashMap<&str, &str> = data
229             .events
230             .iter()
231             .filter(|e| e.kind == "PushEvent")
232             .filter_map(|e| {
233                 let sha = e.payload["head"].as_str()?;

```

```

234         let branch = e.payload["ref"].as_str()?.trim_start_matches("refs/heads/");
235         Some((sha, branch))
236     })
237     .collect();
238
239     let bold = builder.font(true, false).clone();
240     let black = printpdf::Color::Rgb(printpdf::Rgb::new(0.0, 0.0, 0.0, None));
241     builder.write_centered("Recent Commits", &bold, printpdf::Pt(16.0), black);
242     builder.vertical_space(12.0);
243     data.commit_details.iter().for_each(|(repo, detail)| {
244         let branch = sha_to_branch.get(detail.sha.as_str()).copied();
245         pdf::diff::render_commit(&mut builder, detail, repo, branch, config.font_size as f32);
246     });
247 }
248
249 let pages = builder.finish();
250 let page_count = pages.len();
251 doc.with_pages(pages);
252 Ok((doc, page_count))
253 }
254
255 // — Helpers —————
256
257 /// Keep only the first PushEvent per (date, repo, branch) – GitHub emits one per push, so a busy
258 /// day can produce many identical-looking entries. Keeping the first (newest) is sufficient.
259 fn coalesce_push_events(events: Vec<GitHubEvent>) -> Vec<GitHubEvent> {
260     let mut seen = std::collections::HashSet::new();
261     events
262         .into_iter()
263         .filter(|event| {
264             if event.kind != "PushEvent" {
265                 return true;
266             }
267             let date = event.created_at.get(..10).unwrap_or(&event.created_at);
268             let branch = event.payload["ref"].as_str().unwrap_or("");
269             seen.insert((
270                 date.to_string(),
271                 event.repo.name.clone(),
272                 branch.to_string(),
273             ))
274         })
275         .collect()
276 }
277
278 fn render_repos_section(
279     builder: &mut crate::pdf::layout::PageBuilder,
280     title: &str,
281     repos: &[GitHubRepo],
282     limit: usize,
283     events: &[GitHubEvent],
284     commit_msgs: &std::collections::HashMap<String, String>,
285 ) {
286     if limit == 0 || repos.is_empty() {
287         return;
288     }
289     let capped: Vec<_> = repos.iter().take(limit).cloned().collect();
290     pdf::user_repos::render(builder, title, &capped, events, commit_msgs);
291 }
292
293 fn format_size(bytes: u64) -> String {
294     if bytes < 1024 {
295         format!("{bytes} B")
296     } else if bytes < 1024 * 1024 {
297         format!("{:.1} KB", bytes as f64 / 1024.0)
298     } else {
299         format!("{:.1} MB", bytes as f64 / (1024.0 * 1024.0))
300     }
301 }
302
303 fn elapsed_str(d: std::time::Duration) -> String {
304     if d.as_millis() < 1000 {
305         format!("{ms}ms", d.as_millis())
306     } else {
307         format!("{:.1}s", d.as_secs_f64())
308     }
309 }
310
311 #[cfg(test)]

```

```
312 mod tests {
313     use super::*;
314     use crate::github::{CommitAuthor, CommitFile, CommitInfo, EventRepo, GitHubUser};
315     use crate::types::{ActivityFilter, PaperSize};
316
317     fn make_push_event(repo: &str) -> GitHubEvent {
318         GitHubEvent {
319             kind: "PushEvent".to_string(),
320             repo: EventRepo {
321                 name: repo.to_string(),
322             },
323             payload: serde_json::json!({ "ref": "refs/heads/main", "commits": [] }),
324             created_at: "2024-03-01T12:00:00Z".to_string(),
325         }
326     }
327
328     #[test]
329     fn format_size_bytes() {
330         assert_eq!(super::format_size(0), "0 B");
331         assert_eq!(super::format_size(1023), "1023 B");
332     }
333
334     #[test]
335     fn format_size_kilobytes() {
336         assert_eq!(super::format_size(1024), "1.0 KB");
337     }
338
339     #[test]
340     fn format_size_megabytes() {
341         assert_eq!(super::format_size(1024 * 1024), "1.0 MB");
342     }
343
344     #[test]
345     fn elapsed_str_milliseconds() {
346         assert_eq!(
347             super::elapsed_str(std::time::Duration::from_millis(42)),
348             "42ms"
349         );
350         assert_eq!(
351             super::elapsed_str(std::time::Duration::from_millis(999)),
352             "999ms"
353         );
354     }
355
356     #[test]
357     fn elapsed_str_seconds() {
358         assert_eq!(
359             super::elapsed_str(std::time::Duration::from_millis(1500)),
360             "1.5s"
361         );
362     }
363
364     #[test]
365     fn render_repos_section_empty_is_noop() {
366         let mut doc = printpdf::PdfDocument::new("test");
367         let fonts = crate::pdf::fonts::load_fonts(&mut doc).unwrap();
368         let uc = mock_config(0);
369         let mut builder = crate::pdf::create_user_builder(&uc, fonts);
370         let page_before = builder.current_page();
371         super::render_repos_section(
372             &mut builder,
373             "title",
374             &[],
375             5,
376             &[],
377             &std::collections::HashMap::new(),
378         );
379         assert_eq!(builder.current_page(), page_before);
380     }
381
382     #[test]
383     fn render_repos_section_zero_limit_is_noop() {
384         let mut doc = printpdf::PdfDocument::new("test");
385         let fonts = crate::pdf::fonts::load_fonts(&mut doc).unwrap();
386         let uc = mock_config(0);
387         let mut builder = crate::pdf::create_user_builder(&uc, fonts);
388         let page_before = builder.current_page();
389         super::render_repos_section(
```

```
390     &mut builder,
391     "title",
392     &[crate::github::GitHubRepo {
393         name: "x".into(),
394         full_name: "a/x".into(),
395         html_url: "https://github.com/a/x".into(),
396         description: None,
397         language: None,
398         stargazers_count: 0,
399         forks_count: 0,
400         open_issues_count: 0,
401         size: 0,
402         pushed_at: None,
403         updated_at: None,
404         created_at: None,
405         fork: false,
406     }],
407     0,
408     &[],
409     &std::collections::HashMap::new(),
410 );
411 assert_eq!(builder.current_page(), page_before);
412 }
413
414 #[test]
415 fn coalesce_keeps_first_push_per_day_branch() {
416     let events = vec![
417         make_push_event("alice/a"),
418         make_push_event("alice/a"),
419         make_push_event("alice/a"),
420     ];
421     let out = coalesce_push_events(events);
422     assert_eq!(out.len(), 1);
423 }
424
425 #[test]
426 fn coalesce_keeps_different_branches_separate() {
427     let mut ev2 = make_push_event("alice/a");
428     ev2.payload["ref"] = serde_json::json!("refs/heads/dev");
429     let events = vec![make_push_event("alice/a"), ev2];
430     assert_eq!(coalesce_push_events(events).len(), 2);
431 }
432
433 #[test]
434 fn coalesce_preserves_non_push_events() {
435     let events = vec![
436         GitHubEvent {
437             kind: "WatchEvent".to_string(),
438             repo: EventRepo {
439                 name: "alice/a".to_string(),
440             },
441             payload: serde_json::json!({}),
442             created_at: "2024-03-01T00:00:00Z".to_string(),
443         },
444         make_push_event("alice/a"),
445         make_push_event("alice/a"),
446     ];
447     let out = coalesce_push_events(events);
448     assert_eq!(out.len(), 2);
449     assert_eq!(out[0].kind, "WatchEvent");
450 }
451
452 // — render_to_doc offline tests —————
453
454 fn mock_user() -> GitHubUser {
455     GitHubUser {
456         login: "alice".to_string(),
457         name: Some("Alice".to_string()),
458         bio: None,
459         location: None,
460         company: None,
461         blog: None,
462         email: None,
463         public_repos: 5,
464         followers: 10,
465         following: 5,
466         created_at: "2020-01-01T00:00:00Z".to_string(),
467         html_url: "https://github.com/alice".to_string(),
```

```

468     }
469 }
470
471 fn mock_config(commits: usize) -> UserReportConfig {
472     UserReportConfig {
473         username: "alice".to_string(),
474         output_path: "/tmp/test-commits.pdf".into(),
475         paper_size: PaperSize::A4,
476         landscape: false,
477         last_repos: 0,
478         last_commits: commits,
479         no_diffs: false,
480         font_size: 8.0,
481         github_token: None,
482         since: None,
483         until: None,
484         activity: ActivityFilter::All,
485         events: 0,
486     }
487 }
488
489 /// Generate a mock `(repo, CommitDetail)` with a large diff patch so it
490 /// occupies several lines in the rendered PDF.
491 fn mock_commit_detail(idx: usize) -> (String, CommitDetail) {
492     let patch: String = (0..80).map(|i| format!("+added line {i}\n")).collect();
493     (
494         format!("alice/repo{idx}"),
495         CommitDetail {
496             sha: format!("{idx:040x}"),
497             html_url: format!("https://github.com/alice/repo{idx}/commit/{idx:040x}"),
498             commit: CommitInfo {
499                 message: format!("commit #{idx}: add many lines"),
500                 author: CommitAuthor {
501                     name: "Alice".to_string(),
502                     date: format!("2024-03-{:02}T12:00:00Z", idx % 28 + 1),
503                 },
504             },
505             files: vec![CommitFile {
506                 filename: format!("src/module{idx}.rs"),
507                 status: "modified".to_string(),
508                 additions: 80,
509                 deletions: 0,
510                 patch: Some(patch),
511             }],
512         },
513     )
514 }
515
516 fn empty_report_data() -> UserReportData {
517     UserReportData {
518         user: mock_user(),
519         total_stars: 0,
520         starred_repos: vec![],
521         active_repos: vec![],
522         pushed_repos: vec![],
523         events: vec![],
524         commit_msgs: std::collections::HashMap::new(),
525         commit_details: vec![],
526     }
527 }
528
529 #[test]
530 fn render_to_doc_no_commits_succeeds() {
531     let (_, pages) = render_to_doc(&mock_config(0), &empty_report_data()).unwrap();
532     assert!(pages > 0);
533 }
534
535 /// More commits with large diffs must produce more PDF pages than zero commits.
536 /// This verifies the `--last-commits` flag actually drives the diff render path.
537 #[test]
538 fn more_commits_yields_more_pages() {
539     let (_, pages_baseline) = render_to_doc(&mock_config(0), &empty_report_data()).unwrap();
540
541     let data_with_commits = UserReportData {
542         commit_details: (0..10).map(mock_commit_detail).collect(),
543         ..empty_report_data()
544     };
545     let (_, pages_with_commits) = render_to_doc(&mock_config(10), &data_with_commits).unwrap();

```

```
546
547     assert!(
548         pages_with_commits > pages_baseline,
549         "expected more pages with commits ({pages_with_commits}) than without ({pages_baseline})"
550     );
551 }
552 }
```

## tests/integration.rs

```

1 use std::path::{Path, PathBuf};
2
3 use tempfile::TempDir;
4
5 use gitprint::types::{Config, PaperSize};
6
7 async fn git_in(dir: &str, args: &[&str]) {
8     let output = tokio::process::Command::new("git")
9         .args(["-C", dir])
10        .args(args)
11        .output()
12        .await
13        .unwrap();
14    assert!(
15        output.status.success(),
16        "git {:?} failed: {}",
17        args,
18        String::from_utf8_lossy(&output.stderr)
19    );
20 }
21
22 async fn create_test_repo() -> TempDir {
23     let dir = TempDir::new().unwrap();
24     let p = dir.path().to_str().unwrap().to_string();
25
26     git_in(&p, &["init", "-b", "main"]).await;
27
28     // git config writes must be sequential (both modify .git/config, git's file lock
29     // is advisory and concurrent writes fail). File writes are independent so they
30     // run in parallel with the sequential git config block.
31     tokio::join!(
32         async {
33             git_in(&p, &["config", "user.email", "test@test.com"]).await;
34             git_in(&p, &["config", "user.name", "Test"]).await;
35         },
36         async {
37             tokio::try_join!(
38                 tokio::fs::write(
39                     dir.path().join("main.rs"),
40                     "fn main() {\n    println!(\"hello\");\n}\n",
41                 ),
42                 tokio::fs::write(
43                     dir.path().join("lib.rs"),
44                     "pub fn add(a: i32, b: i32) -> i32 {\n    a + b\n}\n",
45                 ),
46                 tokio::fs::write(dir.path().join("README.md"), "# Test Repo\n"),
47                 tokio::fs::create_dir_all(dir.path().join("src")),
48             )
49             .unwrap();
50             // src/ now exists; write util.rs after create_dir_all completes.
51             tokio::fs::write(
52                 dir.path().join("src/util.rs"),
53                 "// utility\npub fn noop() {}\n",
54             )
55             .await
56             .unwrap();
57         },
58     );
59
60     git_in(&p, &["add", "."]).await;
61     git_in(&p, &["commit", "-m", "initial commit"]).await;
62
63     dir
64 }
65
66 fn test_config(repo_path: PathBuf, output_path: PathBuf) -> Config {
67     Config {
68         repo_path,
69         output_path,
70         include_patterns: vec![],
71         exclude_patterns: vec![],
72         theme: "InspiredGitHub".to_string(),
73         font_size: 8.0,
74         no_line_numbers: false,
75         toc: true,
76         file_tree: true,
77         branch: None,

```

```
78     commit: None,
79     paper_size: PaperSize::A4,
80     landscape: false,
81     remote_url: None,
82 }
83 }
84
85 // — git module tests —————
86
87 #[tokio::test]
88 async fn git_verify_repo_valid() -> Result<(), Box<dyn std::error::Error>> {
89     let repo = create_test_repo().await;
90     let info = gitprint::git::verify_repo(repo.path()).await?;
91     assert!(info.is_git);
92     assert!(info.scope.is_none());
93     assert!(info.single_file.is_none());
94     Ok(())
95 }
96
97 #[tokio::test]
98 async fn git_verify_repo_subdir() -> Result<(), Box<dyn std::error::Error>> {
99     let repo = create_test_repo().await;
100    let info = gitprint::git::verify_repo(&repo.path().join("src")).await?;
101    assert!(info.is_git);
102    assert_eq!(info.scope, Some(PathBuf::from("src")));
103    assert!(info.single_file.is_none());
104    Ok(())
105 }
106
107 #[tokio::test]
108 async fn git_verify_repo_single_file_in_git() -> Result<(), Box<dyn std::error::Error>> {
109    let repo = create_test_repo().await;
110    let info = gitprint::git::verify_repo(&repo.path().join("main.rs")).await?;
111    assert!(info.is_git);
112    assert_eq!(info.single_file, Some(PathBuf::from("main.rs")));
113    assert!(info.scope.is_none());
114    Ok(())
115 }
116
117 #[tokio::test]
118 async fn git_verify_repo_plain_directory() -> Result<(), Box<dyn std::error::Error>> {
119    let dir = TempDir::new()?;
120    let info = gitprint::git::verify_repo(dir.path()).await?;
121    assert(!info.is_git);
122    assert!(info.single_file.is_none());
123    Ok(())
124 }
125
126 #[tokio::test]
127 async fn git_verify_repo_plain_file() -> Result<(), Box<dyn std::error::Error>> {
128    let dir = TempDir::new()?;
129    tokio::fs::write(dir.path().join("hello.rs"), "fn main() {}")
130        .await
131        .unwrap();
132    let info = gitprint::git::verify_repo(&dir.path().join("hello.rs")).await?;
133    assert(!info.is_git);
134    assert_eq!(info.single_file, Some(PathBuf::from("hello.rs")));
135    Ok(())
136 }
137
138 #[tokio::test]
139 async fn git_verify_repo_nonexistent_path() {
140    assert!(
141        gitprint::git::verify_repo(Path::new("/nonexistent/path"))
142            .await
143            .is_err()
144    );
145 }
146
147 #[tokio::test]
148 async fn git_get_metadata() -> Result<(), Box<dyn std::error::Error>> {
149    let repo = create_test_repo().await;
150    let config = test_config(repo.path().to_path_buf(), PathBuf::from("/tmp/test.pdf"));
151    let metadata = gitprint::git::get_metadata(repo.path(), &config, true, None).await?;
152
153    assert(!metadata.name.is_empty());
154    assert_eq!(metadata.branch, "main");
155    assert_eq!(metadata.commit_hash.len(), 40);

```

```
156     assert!(metadata.commit_hash.chars().all(|c| c.is_ascii_hexdigit()));
157     assert_eq!(metadata.commit_hash_short.len(), 7);
158     assert_eq!(metadata.commit_message, "initial commit");
159     assert!(metadata.commit_date.is_empty());
160     Ok(())
161 }
162
163 #[tokio::test]
164 async fn git_get_metadata_plain_directory() -> Result<(), Box<dyn std::error::Error>> {
165     let dir = TempDir::new()?;
166     let config = test_config(dir.path().to_path_buf(), PathBuf::from("/tmp/test.pdf"));
167     let metadata = gitprint::git::get_metadata(dir.path(), &config, false, None).await?;
168
169     assert!(metadata.name.is_empty());
170     assert!(metadata.branch.is_empty());
171     assert!(metadata.commit_hash.is_empty());
172     assert!(metadata.commit_date.is_empty());
173     Ok(())
174 }
175
176 #[tokio::test]
177 async fn git_get_metadata_with_branch() -> Result<(), Box<dyn std::error::Error>> {
178     let repo = create_test_repo().await;
179     let mut config = test_config(repo.path().to_path_buf(), PathBuf::from("/tmp/test.pdf"));
180     config.branch = Some("main".to_string());
181     let metadata = gitprint::git::get_metadata(repo.path(), &config, true, None).await?;
182     assert_eq!(metadata.branch, "main");
183     Ok(())
184 }
185
186 #[tokio::test]
187 async fn git_list_tracked_files() -> Result<(), Box<dyn std::error::Error>> {
188     let repo = create_test_repo().await;
189     let config = test_config(repo.path().to_path_buf(), PathBuf::from("/tmp/test.pdf"));
190     let files = gitprint::git::list_tracked_files(repo.path(), &config, true, None).await?;
191
192     assert!(files.contains(&PathBuf::from("main.rs")));
193     assert!(files.contains(&PathBuf::from("lib.rs")));
194     assert!(files.contains(&PathBuf::from("src/util.rs")));
195     assert!(files.contains(&PathBuf::from("README.md")));
196     assert_eq!(files.len(), 4);
197     Ok(())
198 }
199
200 #[tokio::test]
201 async fn git_list_files_plain_directory() -> Result<(), Box<dyn std::error::Error>> {
202     let dir = TempDir::new()?;
203     tokio::try_join!(
204         tokio::fs::write(dir.path().join("hello.rs"), "fn main() {}"),
205         tokio::fs::create_dir(dir.path().join("sub")),
206     )
207     .unwrap();
208     tokio::fs::write(dir.path().join("sub/world.rs"), "pub fn world() {}")
209     .await
210     .unwrap();
211     let config = test_config(dir.path().to_path_buf(), PathBuf::from("/tmp/test.pdf"));
212     let files = gitprint::git::list_tracked_files(dir.path(), &config, false, None).await?;
213
214     assert!(files.contains(&PathBuf::from("hello.rs")));
215     assert!(files.contains(&PathBuf::from("sub/world.rs")));
216     assert_eq!(files.len(), 2);
217     Ok(())
218 }
219
220 #[tokio::test]
221 async fn git_read_file_content() -> Result<(), Box<dyn std::error::Error>> {
222     let repo = create_test_repo().await;
223     let config = test_config(repo.path().to_path_buf(), PathBuf::from("/tmp/test.pdf"));
224     let content =
225         gitprint::git::read_file_content(repo.path(), Path::new("main.rs"), &config).await?;
226
227     assert!(content.contains("fn main()"));
228     assert!(content.contains("println!"));
229     Ok(())
230 }
231
232 #[tokio::test]
233 async fn git_read_file_content_nonexistent() {
```

```
234     let repo = create_test_repo().await;
235     let config = test_config(repo.path().to_path_buf(), PathBuf::from("/tmp/test.pdf"));
236     let result =
237         gitprint::git::read_file_content(repo.path(), Path::new("nonexistent.rs"), &config).await;
238     assert!(result.is_err());
239 }
240
241 // — full pipeline tests —————
242
243 #[tokio::test]
244 async fn full_pipeline() -> Result<(), Box<dyn std::error::Error>> {
245     let repo = create_test_repo().await;
246     let out_dir = TempDir::new()?;
247     let output_path = out_dir.path().join("output.pdf");
248     let config = test_config(repo.path().to_path_buf(), output_path.clone());
249
250     gitprint::run(&config).await?;
251
252     assert!(output_path.exists());
253     assert!(std::fs::metadata(&output_path)?.len() > 0);
254     Ok(())
255 }
256
257 #[tokio::test]
258 async fn full_pipeline_with_include_filter() -> Result<(), Box<dyn std::error::Error>> {
259     let repo = create_test_repo().await;
260     let out_dir = TempDir::new()?;
261     let output_path = out_dir.path().join("output.pdf");
262     let mut config = test_config(repo.path().to_path_buf(), output_path.clone());
263     config.include_patterns = vec!["*.rs".to_string()];
264
265     gitprint::run(&config).await?;
266
267     assert!(output_path.exists());
268     assert!(std::fs::metadata(&output_path)?.len() > 0);
269     Ok(())
270 }
271
272 #[tokio::test]
273 async fn full_pipeline_with_exclude_filter() -> Result<(), Box<dyn std::error::Error>> {
274     let repo = create_test_repo().await;
275     let out_dir = TempDir::new()?;
276     let output_path = out_dir.path().join("output.pdf");
277     let mut config = test_config(repo.path().to_path_buf(), output_path.clone());
278     config.exclude_patterns = vec!["*.md".to_string()];
279
280     gitprint::run(&config).await?;
281     assert!(output_path.exists());
282     Ok(())
283 }
284
285 #[tokio::test]
286 async fn full_pipeline_no_toc_no_tree() -> Result<(), Box<dyn std::error::Error>> {
287     let repo = create_test_repo().await;
288     let out_dir = TempDir::new()?;
289     let output_path = out_dir.path().join("output.pdf");
290     let mut config = test_config(repo.path().to_path_buf(), output_path.clone());
291     config.toc = false;
292     config.file_tree = false;
293
294     gitprint::run(&config).await?;
295     assert!(output_path.exists());
296     Ok(())
297 }
298
299 #[tokio::test]
300 async fn full_pipeline_no_line_numbers() -> Result<(), Box<dyn std::error::Error>> {
301     let repo = create_test_repo().await;
302     let out_dir = TempDir::new()?;
303     let output_path = out_dir.path().join("output.pdf");
304     let mut config = test_config(repo.path().to_path_buf(), output_path.clone());
305     config.no_line_numbers = true;
306
307     gitprint::run(&config).await?;
308     assert!(output_path.exists());
309     Ok(())
310 }
311
```

```
312 #[tokio::test]
313 async fn full_pipeline_landscape() -> Result<(), Box<dyn std::error::Error>> {
314     let repo = create_test_repo().await;
315     let out_dir = TempDir::new()?;
316     let output_path = out_dir.path().join("output.pdf");
317     let mut config = test_config(repo.path().to_path_buf(), output_path.clone());
318     config.landscape = true;
319
320     gitprint::run(&config).await?;
321     assert!(output_path.exists());
322     Ok(())
323 }
324
325 #[tokio::test]
326 async fn full_pipeline_letter_paper() -> Result<(), Box<dyn std::error::Error>> {
327     let repo = create_test_repo().await;
328     let out_dir = TempDir::new()?;
329     let output_path = out_dir.path().join("output.pdf");
330     let mut config = test_config(repo.path().to_path_buf(), output_path.clone());
331     config.paper_size = PaperSize::Letter;
332
333     gitprint::run(&config).await?;
334     assert!(output_path.exists());
335     Ok(())
336 }
337
338 #[tokio::test]
339 async fn full_pipeline_subdir() -> Result<(), Box<dyn std::error::Error>> {
340     let repo = create_test_repo().await;
341     let out_dir = TempDir::new()?;
342     let output_path = out_dir.path().join("output.pdf");
343     let config = test_config(repo.path().join("src"), output_path.clone());
344
345     gitprint::run(&config).await?;
346
347     assert!(output_path.exists());
348     assert!(std::fs::metadata(&output_path)?.len() > 0);
349     Ok(())
350 }
351
352 #[tokio::test]
353 async fn full_pipeline_single_file() -> Result<(), Box<dyn std::error::Error>> {
354     let repo = create_test_repo().await;
355     let out_dir = TempDir::new()?;
356     let output_path = out_dir.path().join("output.pdf");
357     let config = test_config(repo.path().join("main.rs"), output_path.clone());
358
359     gitprint::run(&config).await?;
360
361     assert!(output_path.exists());
362     assert!(std::fs::metadata(&output_path)?.len() > 0);
363     Ok(())
364 }
365
366 #[tokio::test]
367 async fn full_pipeline_plain_directory() -> Result<(), Box<dyn std::error::Error>> {
368     let dir = TempDir::new()?;
369     tokio::try_join!(
370         tokio::fs::write(dir.path().join("main.rs"), "fn main() {}\\n"),
371         tokio::fs::write(
372             dir.path().join("lib.rs"),
373             "pub fn add(a: i32, b: i32) -> i32 { a + b }\\n",
374         ),
375     )
376     .unwrap();
377     let out_dir = TempDir::new()?;
378     let output_path = out_dir.path().join("output.pdf");
379     let config = test_config(dir.path().to_path_buf(), output_path.clone());
380
381     gitprint::run(&config).await?;
382
383     assert!(output_path.exists());
384     assert!(std::fs::metadata(&output_path)?.len() > 0);
385     Ok(())
386 }
387
388 #[tokio::test]
389 async fn full_pipeline_nonexistent_repo() {
```

```
390     let out_dir = TempDir::new().unwrap();
391     let output_path = out_dir.path().join("output.pdf");
392     let config = test_config(PathBuf::from("/nonexistent/repo"), output_path);
393
394     assert!(gitprint::run(&config).await.is_err());
395 }
396
397 #[tokio::test]
398 async fn full_pipeline_invalid_theme() {
399     let repo = create_test_repo().await;
400     let out_dir = TempDir::new().unwrap();
401     let output_path = out_dir.path().join("output.pdf");
402     let mut config = test_config(repo.path().to_path_buf(), output_path);
403     config.theme = "NonExistentTheme".to_string();
404
405     let err = gitprint::run(&config).await.unwrap_err();
406     assert!(err.to_string().contains("NonExistentTheme"));
407     assert!(err.to_string().contains("--list-themes"));
408 }
409
410 #[tokio::test]
411 async fn full_pipeline_include_excludes_everything() -> Result<(), Box<dyn std::error::Error>> {
412     let repo = create_test_repo().await;
413     let out_dir = TempDir::new()?;
414     let output_path = out_dir.path().join("output.pdf");
415     let mut config = test_config(repo.path().to_path_buf(), output_path.clone());
416     config.include_patterns = vec!["*.nonexistent".to_string()];
417
418     gitprint::run(&config).await?;
419     assert!(output_path.exists());
420     Ok(())
421 }
422
423 #[tokio::test]
424 async fn full_pipeline_custom_font_size() -> Result<(), Box<dyn std::error::Error>> {
425     let repo = create_test_repo().await;
426     let out_dir = TempDir::new()?;
427     let output_path = out_dir.path().join("output.pdf");
428     let mut config = test_config(repo.path().to_path_buf(), output_path.clone());
429     config.font_size = 12.0;
430
431     gitprint::run(&config).await?;
432     assert!(output_path.exists());
433     Ok(())
434 }
```