Bricked-Up Specification: Backend

Rumen Mitov

February 13, 2025

Contents

1 Productivity Scheme

Each member of the team is expected to work 10 hours per week. The member should record what they are working on, and how long they worked on it.

Each member will have the freedom to choose a feature / task to work on (unless there are some urgent tasks). To start working on a task, the member must first create a GitHub issue describing what the task entails. The member then branches off of the upstream branch to work on the task. Once the task is finished, the member submits a Pull Request (aka PR) to the `dev` branch and requests a PR Review from the backend lead. The PR will need to adhere to the criteria specified in PR Approval Criteria in order to be approved.

Whoever creates the feature is responsible for maintaining it. We will do this to exploit the maintainers familiarity with the code, thus saving time.

By this we aim to increase the concurrency with which our team operates. Status updates will be discussed during weekly team meetings in order to facilitate discussion and to keep the bigger picture in mind.

Once a week, the backend lead will merge `dev` to `master`.

1.1 PR Approval Criteria

Here are the criteria the PR must adhere to in order to be accepted:

- No merge conflicts
- All tests are passing
- New features **must** come with unit tests
- No junk files are present (e.g. `.vscode`)
- Directory hierarchy is followed
- PR commit message is properly structured

1.2 Commit Message Prefix

This applies to final commits (i.e. commits when merging to `dev`):

- **feat(<feature>, close <github issue nr.>)**: description
- **refactor(<feature>, close <github issue nr.>)**: description

- **fix(<feature>, close <github issue nr.>):** description
- **hotfix(<bug>, close <github issue nr.>)**: description
- **bug(<bug>, close <github issue nr.>)**: description
- **tests(<feature>, close <github issue nr.>)**: description
- **misc**: description
- **docs**: description
- **update**: description

```
feat(login, close #13): created login endpoint
```

2 System Requirements

We will be utilizing a traditional client-server architecture with a centralized database (as opposed to a distributed one), we will require a Virtual Private Server. The state of the project is still an MVP, hence we are not expecting a lot of users. Thus, the server does not need to be powerful; we prioritize cost over performance. A cheap configuration from a provider like Hetzner should suffice:

Plan	vCPU	RAM	NVMe	Traffic	IPv4
CX22	2	4GB	40GB	20TB	Yes

For a domain name we can use a free domain name provider with Let's Encrypt for a free SSL certification.

We will be using Github for project management and version management. We will use WhatsApp as a communication channel and Microsoft Teams for communicating with our customer.

3 Software Requirements

3.1 Dependability

The backend should be resilient and avoid crashes.

3.2 Security and Secrecy

Sensitive information should be encrypted on the server and can only be accessed by properly authenticated users.

3.3 Performance on a Low-Scale

The backend should be capable of handling relatively low-workloads without outages and lagging.

4 Architecture

4.1 Programming Language: Golang

As a garbage-collected, system's programming language Golang has been proven to work exceptionally well in the industry as a backend language for many services. Additionally, it's simple syntax and extensive standard library made it an attractive option for our team's skill set.

Moreover, the developer tooling for the language is exceptional, allowing for an ergonomic developer experience when it comes to development, documentation, and testing.

4.2 Database: Sqlite

Sqlite is used extensively in the tech industry. It is extremely light-weight and simple to work with (due to the entire database being contained in a single file). It is a relational-database which fits perfectly with modeling a Project-Management system.

4.3 Client-Server Authentication

Authentication between the client and the server will be done through session tokens which will be saved in our database. We decided that session tokens will be more appropriate than JWT, since our service is consolidated into one centralized service.

4.4 Containerization: Podman

We will use the Podman to containerize our backend, due to Podman's rootless capabilities, open source nature and Kubernetes-like offerings. We will use podman's virtual bridging to isolate the database container entirely and partially isolate the Golang application.

Here is how the backend will be organized:

1. The entire backend will be organized in a pod called `backend-pod` with a network `backend-pod`

2. The Golang backend will be a container called `backend-prod` with exposed port `3100:443`

3. The Sqlite database will be a container called `db-prod` and it will **NOT** be exposed to the host

4.5 Reverse-Proxy: Nginx

We will use Nginx as a reverse-proxy due to its ease-of-configuration and performance. Assuming that our domain name is `<brickedup>` this will be the hierarchy of our sub-domains:

Sub-Domain	Localhost Port	Description
`home.<brickedup>`	80,443	frontend's index
`backend.<brickedup>`	3100	backend's Golang application
`docs.backend.<brickedup>`	6060	backend's docs
`cicd.backend.<brickedup>`	7123	CI/CD for the backend

5 Database Design

The production database file will be called: `bricked-up_prod.db` and it will be located in the root of the repo (will live only on the server, all *.db files will be ignored by `.gitignore`).

```
---
config:
  theme: dark
title: Bricked-Up ER-Diagram
---
erDiagram
    TEAM {
        int id PK
        string name UK
    }

    USER {
        int id PK
        string name
    }
```

```
PROJECT {
    int id PK
    int teamid FK
    string name UK
}

ISSUE {
    int id PK
    string title
    string desc
    int tagid FK
}

TAG {
    int id PK
    int projectid FK
    string name UK
}


USER |o--o{ USERS_IN_TEAMS : in
TEAM ||--|{ USERS_IN_TEAMS :has
USERS_IN_TEAMS {
    int id PK
    int userid FK
    int teamid FK
}

TEAM ||--o{ TEAM_PROJECTS : has
PROJECT ||--o| TEAM_PROJECTS : belongs_to
TEAM_PROJECTS {
    int id PK
    int teamid FK
    int projectid FK
}

PROJECT ||--o{ PROJECT_ISSUES : has
ISSUE ||--|| PROJECT_ISSUES : belongs_to
PROJECT_ISSUES {
    int id PK
```

```
    int projectid FK
    int issueid FK
}

USER ||--o{ USER_ISSUES : responsible_for
ISSUE ||--o{ USER_ISSUES : assigned_to
USER_ISSUES {
    int id PK
    int userid FK
    int issueid FK
}

ISSUE ||--o| TAG : has

PROJECT ||--o{ TAG : offers
```

Figure 1: A Mermaid.js ER Diagram describing our database schema:

6 Documentation

We will use godoc to generate documentation. This will be hosted on `localhost:6060`.

7 Testing

We will use Go's testing package.

7.1 Unit Tests

All features should have unit tests. A demo database will be populated from an SQL script with dummy data. The database file should be called `bricked-up_test.db` and it should be located in the root of the repository.

7.2 TODO Integration Tests

7.3 TODO Fuzzing Tests

8 Deployment

The following describes the full deployment pipeline (assuming task is complete and ready to be pushed upstream):

1. All tests run successfully
2. PR is submitted (tests are run on Github Actions to ensure everything works)
3. PR is reviewed by lead (must be accepted to continue)
4. Lead **squash merges** PR into `dev` branch (task issue is closed)
5. `dev` is **squash merged** to `master` branch (happens once a week)
6. once a change has been pushed to `master`, a webhook is sent to our server's CI/CD
7. CI/CD program pulls changes and rebuilds backend on the server