Bricked-Up Specification: Backend

Rumen Mitov

March 23, 2025

Contents

1 Productivity Scheme

Each member of the team is expected to complete their tasks for the week as well as attend the general meeting and the backend meeting (each of which will occur once per week). Each team member will have an allocated time-slot in the week where they have to be online in case there are requests for changes in their Pull Request. The tasks will be agreed upon in the backend meeting.

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 `master` 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.

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 `master`):

Commit Type	Issue Type	Github Action	Use Case
feat	<feature>	close <issue nr>	Introduction of a new feature.
refactor	<feature>	close <issue nr>	Changing the implementation of an existing feature.
fix	<feature> / <bug>	close <issue nr>	Fixing a feature or a bug.
tests	<feature>	close <issue nr>	Adding / removing / changing tests.
docs	x	x	Documentation.
misc	x	x	Miscellaneous (renaming of files, removing junk, etc.)

An example merge commit message would be the following:

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

1.3 Creation of Issues

Whenever appropriate use the repository's issue templates. If a heading is irrelevant (i.e. empty) delete it.

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

```
---
config:
theme: dark
title: Bricked-Up Architecture
---
architecture-beta
    service traffic(internet)[internet]

    group vps(server)[vps]
    service nginx(internet)[nginx] in vps
    service sqlite(database)[sqlite] in vps
    service frontend(server)[frontend] in vps
    service cicd(server)[CICD] in vps

    group container(internet)[container] in vps
    service backend(server)[backend] in container

    group docs(internet)[container] in vps
    service backenddocs(server)[docs] in docs
```

```
nginx:L -- R:traffic
nginx:T -- B:backend
nginx:R -- L:frontend
nginx:B -- T:backenddocs
nginx:R -- L:cicd

backend:R -- L:sqlite
```

Figure 1: A Mermaid.js diagram displaying our 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.

We will be using ModernC's Sqlite Driver.

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.

The Golang backend will be a container called `backend-prod` with exposed port `3100:443`. The Sqlite database file (`bricked-up_prod.db`) will be mounted as a volume to `backend-prod`.

Additionally, the backend docs will be in a container called `backend-docs` with exposed port `6060:6060`.

4.5 Reverse-Proxy: Apache

We will use Apache as a reverse-proxy due to its ease-of-configuration and performance. The website can be accessed here: `http://clabsql.clamv.constructor.university/`. The table below maps the ports and path to the service provided.

Port	Path	Description
3000	`/`	frontend's index
3100	`/`	backend router
6060	`/pkg/brickedup/backend/`	backend's docs
7123	`/frontend`	CI/CD for the frontend
7123	`/backend`	CI/CD for the backend

Since we need the CI/CD pipeline to be accessible from devices outside the university's network, the URL `https://webhook.constructor.university/` maps to the internal port `:7123`.

4.6 Repository Organization

Here is the directory hierarchy for the backend repository:

- **src** - for main endpoint handling / routing
- **src/utils** - common utility functions
- **sql** - sql scripts to initiate database tables / populate database with dummy data

When developing, compile the code to a binary in the **bin** directory in the root of the repo. The lead will make sure that the **bin** directory will be ignored by Git, that way we do not push any unnecessary binaries.

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

    %%
    %% Primative Entities
    %%

    ORGANIZATION {
        int id PK
        string name UK
    }

    ORG_ROLE {
        int id PK
        int orgid FK
        string name
        bool can_read
        bool can_write
        bool can_exec
```

```
}

USER {
    int id PK
    int verifyid FK
    string email UK
    string password
    string name
    string avatar UK
    bool verified
}

SESSION {
    int id PK
    int userid FK
    timestamp expires
}

PROJECT {
    int id PK
    int orgid FK
    string name
    int budget
    string charter
    bool archived
}

PROJECT_ROLE {
    int id PK
    int projectid FK
    string name
    bool can_read
    bool can_write
    bool can_exec
}

ISSUE {
    int id PK
    string title
    string desc
    int tagid FK
    int priority
    timestamp created
    timestamp completed
    int cost
}

DEPENDENCY {
    int id PK
    int issueid FK
```

```
    int dependency FK
}

TAG {
    int id PK
    int projectid FK
    string name

    %% color should be stored as a hex value
    int color
}

REMINDER {
    int id PK
    int issueid FK
    int userid FK
}

%%
%% Relationships
%%

%% Verify user
USER ||--o| VERIFY_USER : is
VERIFY_USER {
    int id PK
    int code UK
    timestamp expires
}

%% Password reset
USER ||--o{ PASSRESET : has
PASSRESET {
    int id PK
    int code UK
    timestamp expires
}

%% User login sessions
USER ||--o{ SESSION : has

%% Organization members
USER |o--o{ ORG_MEMBER : is
ORGANIZATION ||--|{ ORG_MEMBER : has
ORG_MEMBER {
    int id PK
    int userid FK
    int orgid FK
}
```

```
%% Organization roles
ORG_MEMBER ||--|{ ORG_MEMBER_ROLE : has
ORGANIZATION ||--|{ ORG_ROLE : offers
ORG_ROLE ||--o{ ORG_MEMBER_ROLE : contains
ORG_MEMBER_ROLE {
    int id PK
    int memberid FK
    int roleid FK
}

%% Organization projects
ORGANIZATION ||--o{ ORG_PROJECTS : has
PROJECT ||--|| ORG_PROJECTS : belongs_to
ORG_PROJECTS {
    int id PK
    int orgid FK
    int projectid FK
}

%% Project members
PROJECT ||--|{ PROJECT_MEMBER : has
PROJECT_MEMBER {
    int id PK
    int userid FK
    int projectid FK
}

%% Project roles
PROJECT ||--|{ PROJECT_ROLE : has
PROJECT_ROLE ||--o{ PROJECT_MEMBER_ROLE : contains
PROJECT_MEMBER ||--|{ PROJECT_MEMBER_ROLE : has
PROJECT_MEMBER_ROLE {
    int id PK
    int memberid FK
    int roleid FK
}

%% Project issues
PROJECT ||--o{ PROJECT_ISSUES : has
ISSUE ||--|| PROJECT_ISSUES : belongs_to
PROJECT_ISSUES {
    int id PK
    int projectid FK
    int issueid FK
}

%% User-assigned issues
USER ||--o{ USER_ISSUES : responsible_for
ISSUE ||--o{ USER_ISSUES : assigned_to
USER_ISSUES {
```

```
        int id PK
        int userid FK
        int issueid FK
    }

    %% Issue dependencies
    ISSUE ||--o{ DEPENDENCY : has

    %% Tags
    ISSUE ||--o| TAG : has
    PROJECT ||--o{ TAG : offers

    %% Reminders
    ISSUE ||--o{ REMINDER : sends
    USER }o--o{ REMINDER : targets
```

Figure 2: A Mermaid.js ER diagram displaying our database schema.

6 Documentation

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

The comments should be written in the following manner if they are to appear in the documentation:

```
// Package packageName provides functionalities related to ...
package packageName

// TypeName represents ...
type TypeName struct {
        FieldName int // Description of FieldName
}

// FunctionName performs ...
func FunctionName(arg1 int, arg2 string) {
}
```

Figure 3: Source

7 Testing

We will use Go's testing package.

7.1 Unit Tests

All features should have unit tests.

7.2 Integration Tests

Integration tests checks if the communication between our services (e.g. our server and our database) is correct. 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. The tests can now run queries on the testing database.

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 `master` branch (task issue is closed)
5. Once a change has been pushed to `master`, a webhook is sent to our server's CI/CD
6. CI/CD program pulls changes and rebuilds backend on the server