masp

make a simple project

Entwicklerhandbuch

Handbuch für die Weiterentwicklung bestehender und Entwicklung neuer Module

Sven Baer

Phillipp Engelke

Alexa Grube

Tim Härtel

Moritz Marquardt

Otto von Guericke Universität Magdeburg - 4. Juli 2019

Inhaltsverzeichnis

1 Vorwort

masp (make a simple project) ist eine Projektmanagement-Anwendung basierend auf modernen Web-Technologien, die es vor allem kleinen Teams ermöglichen soll, Projekte mit kleinstmöglichem Mehraufwand für die Organisation umzusetzen. Die Software soll genutzt werden, um Projekte zu planen und während des laufenden Projektes einen Überblick zu behalten, wo der aktuelle Projektfortschritt liegt.

Für ein Projekt ist die Software so lange im Einsatz, wie das Projekt in der Planung und Entwicklung benötigt. Eine Software-Instanz soll genau ein Projekt umfassen.

Die Software wurde im Rahmen eines Softwareprojekts unseres Bachelorstudiums an der Otto-von-Guericke-Universität entwickelt. In diesem Zeitraum haben wir das Basis-, Nutzer- und Aufgabenmodul entwickelt.

2 Aufsetzen der Entwicklungsumgebung

Disclaimer: Wir wollen niemanden aus seiner gewohnten Entwicklungsumgebung holen! Dies ist nur das Setup mit dem wir in der Entwicklung gearbeitet haben, und für welches die Dokumentation und alle Skripte ausgelegt ist.

2.1 Benötigte Software

- **Go** (golang.org) - Compiler für den Backend-Code
- **Node.js** (nodejs.org) - Compiler für den Frontend-Code
- **gcc** (gcc.gnu.org) (Windows-Version "TDM-GCC" siehe tdm-gcc.tdragon.net) - Compiler für die SQLite-Datenbankbibliothek
- **Git** (git-scm.com) - Versionskontrollsystem
- **Visual Studio Code** (code.visualstudio.com) - Integrierte Entwicklungsumgebung, optional
- **GitKraken** (gitkraken.com) - Grafische Oberfälche für Git, optional

2.1.1 Installation unter Windows

Nach der Installation aller benötigter Software (siehe oben) müssen noch die Umgebungsvariablen gesetzt werden:

Start, "Umgebungsvariablen für dieses Konto bearbeiten" eintippen & mit Enter bestätigen, oben bei Benutzervariabeln folgendes überprüfen:

- Variable "GOPATH" mit einem leeren Verzeichnis als Wert, in dem Go seine Dependencies speichern kann
- Variable "GOROOT" mit Wert der Go-Installation (enthält Unterordner wie "bin", "src" und "lib")
- Variable "Path" enthält folgende Verzeichnisse ("Bearbeiten")
  - `[GCC-Verzeichnis]/bin`
  - `[Node.js-Verzeichnis]` (Enthält node.exe und node_modules)
  - `[GOROOT]/bin`
  - `[GOPATH]/bin`

Danach sollte der Computer neugestartet werden.

2.1.2 Installation unter Linux

Debian-basierte Systeme werden direkt unterstützt, bei anderen Systemen ist die Installation ähnlich.

```
wget https://release.gitkraken.com/linux/gitkraken-amd64.deb -O
  /tmp/gitkraken.deb

sudo dpkg -i /tmp/gitkraken
sudo apt-get -f install
rm -f /tmp/gitkraken.deb

wget -qO - https://gitlab.com/paulcarroty/vscodium-deb-rpm-
  repo/raw/master/pub.gpg | sudo apt-key add -
echo 'deb https://gitlab.com/paulcarroty/vscodium-deb-rpm-repo/raw/repos/debs/
  vscodium main' | sudo tee /etc/apt/sources.list.d/vscodium.list

curl -sL https://deb.nodesource.com/setup_10.x | sudo -E bash -

sudo apt-get install golang-1.12-go nodejs git gcc codium
```

2.1.3 Installation unter macOS

Für die Installation wird **Homebrew** (brew.sh) empfohlen.

```
/usr/bin/ruby -e "$(curl -fsSL
  https://raw.githubusercontent.com/Homebrew/install/master/install)"

brew install go gcc git npm
brew cask install visual-studio-code gitkraken
```

2.2 VS Code einrichten

Unsere IDE besteht aus Visual Studio Code mit folgenden Plugins erweitert:

- Go (`ms-vscode.go`)
- Vetur (`octref.vetur`)
- EditorConfig for VS Code (`editorconfig.editorconfig`)
- TODO Highlight (`wayou.vscode-todo-highlight`)
- Todo Tree (`gruntfuggly.todo-tree`)
- SQLTools (`mtxr.sqltools`)

Mit den folgenden Befehlen können die Plugins aus der Kommandozeile installiert werden (unter Linux muss mit der obigen Installationsanleitung `codium` statt `code` als Befehl verwendet werden):

```
code --install-extension ms-vscode.go \
     --install-extension octref.vetur \
     --install-extension editorconfig.editorconfig \
     --install-extension wayou.vscode-todo-highlight \
     --install-extension gruntfuggly.todo-tree \
     --install-extension mtxr.sqltools
```

2.3 Repository einrichten

Wir brauchen nun einen leeren Ordner, in welchem in einem Terminal folgende Befehle ausgeführt werden:

```
go mod init git.mo-mar.de/masp
git clone https://git.mo-mar.de/masp/masp.git
git clone https://git.mo-mar.de/masp/tasks.git
git clone https://git.mo-mar.de/masp/users.git
git clone https://git.mo-mar.de/masp/.vscode.git
```

Dieser Ordner kann jetzt mit VS Code geöffnet werden, der integrierte Debugger kann zum Start des Programms verwendet werden.

3 Arbeit mit den Repositories

3.1 Fremde Module hinzufügen oder Module entfernen

Module werden in der Datei `masp/build/modules.txt` verwaltet. Hier kann ein Go-Importpfad pro Zeile definiert werden, der dann als Modul geladen wird. Um die Änderungen an dieser Datei anzuwenden, muss in VS Code mit `Ctrl+Shift+P` die Kommandopalette geöffnet werden und dort `Tasks: Run Task` ausgewählt werden. Im darauffolgenden Fenster gibt es die Option “Clean & Install Dependencies”, welche alle in dieser Datei definierten Module installiert und einbindet.

3.2 Erstellen eines neuen Moduls

Um ein neues Modul für masp zu erstellen, muss ein neues Go-Package/-Modul erstellt werden, das vom Go-Compiler gefunden wird.

3.2.1 Backend-Code

Im Hauptordner des Moduls muss eine `module.go`-Datei liegen, die das Modul beim Basismodul registriert und API-Routen definiert:

`module.go`:

```
package example

import (
    "path"
    "runtime"

    "git.mo-mar.de/masp/masp"
    "github.com/coreos/go-semver/semver"
    "github.com/gin-gonic/gin"
)

// Module definiert das Modul
type Module struct{}

// M ist die Instanz des Moduls und muss immer so heißen, damit andere Module
  darauf zugreifen können
var M = Module{}

// m referenziert die API des Basismoduls und ist private, damit kein Modul
  sich als ein anderes ausgeben kann
var m = masp.RegisterModule(&M)

// Info gibt Informationen über das Modul aus (Name, ID, Version, Dateipfad)
func (*Module) Info() (string, string, *semver.Version, string) {
    _, filename, _, _ := runtime.Caller(0)
    return "Example Module", "example", semver.New("1.0.0"),
  path.Dir(filename)
}

// Routes richtet die vom Modul benötigten Routen ein
func (*Module) Routes(router *gin.RouterGroup) {
      // Hier GIN-Routen auf den Router definieren.
      // Diese haben die Modul-ID (hier "example") als Pfad-Prefix, z.B.
  /api/example/helloworld
      // Für mehr Informationen siehe https://godoc.org/github.com/gin-
  gonic/gin
      router.GET("helloworld", helloWorldHandler)
}

func helloWorldHandler(c *gin.Context) {
    c.String(200, "Hello World\n")
}
```

3.2.2 Frontend-Code

Für das Frontend muss ein Ordner namens `frontend` angelegt werden. In diesem muss es eine `module.vue` geben, die beispielsweise wie folgt aufgebaut sein kann:

`module.vue`:

```
<script>
export default {
  // Frontend-Routen für den Vue Router (https://router.vuejs.org/) definieren
  // Als Konvention sollten Pfade mit /[module-id] beginnen, einzelne
  Resourcen sollten unter /[module-id]/[global-id] verfügbar sein.
  routes() {
    return [{ path: "/hello", component: this }];
  },
  // Einstellungen des Moduls, die in den globalen Projekteinstellungen
  angezeigt werden.
  settings() {
    return {
      module: "example",
      settings: [
        {
            title: "Hallo Welt",
            type: "text",
            name: "hello-world"
        },
        {
            title: "Knopf",
            type: "button",
            name: "anonymous-access"
        },
        {
            title: "Checkbox",
            type: "checkbox",
            name: "unnamed"
        }
      ]
    };
  }
};
</script>
```

Die REST-API kann mit `const res = await api.POST("/api/", { "key": "value" }, { "X-My-Awesome-Header": "totally-awesome" })` angefragt werden, dabei wird ein Fetch-Wrapper (go.momar.io/fetch-wrapper) verwendet - in `res.content` steht damit die Antwort des Servers, ansonsten ist `res` eine normale Response (https://developer.mozilla.org/en-US/docs/Web/API/Response).

3.2.3 Datenbank

Die Datenbank kann im Ordner `database` des Moduls verwaltet werden - hier können SQL-Dateien (empfohlenerweise mit dem Dateiformat `000_first_migration.sql`) erstellt werden, die mit **Goose** (github.com/pressly/goose zur Migration verwendet werden. Eine solche Datei könnte beispielsweise wie folgt aussehen:

```
-- +goose Up
-- Namen von Datenbanktabellen sollten stets mit der Modul-ID beginnen
CREATE TABLE example_mytable (hello TEXT);
-- +goose Down
DROP TABLE example_mytable;
```

Als Datenbank wird SQLite3 verwendet, die Dokumentation dazu ist auf https://sqlite.org/lang.html verfügbar.

3.3 Docker-Image erstellen

Um ein funktionstüchtiges Docker-Image zu erstellen, muss im Ordner `masp` der folgende Befehl ausgeführt werden:

```
docker build -t example/masp -f build/Dockerfile .
```

Dabei ist zu beachten, dass beim Buildvorgang alle Module erneut heruntergeladen werden und der `master`-Branch verwendet wird.

4 Vorhandene Module

Die Dokumentation der REST-API aller grundlegender Module ist als Swagger-Dokumentation auf https://get.mo-mar.de/masp/API-Dokumentation.pdf verfügbar.

4.1 Basismodul

4.1.1 Funktionsumfang

Das Basismodul enthält alle grundlegenden Funktionen einer Webanwendung, die masp-Module ausführen kann - dies beinhaltet Logging, Datenbankzugriff, Verwaltung von globalen IDs, Projekteinstellungen, E-Mail-Versand, sowie das grundlegende Design und Layout des Frontends.

4.1.2 Globale IDs

Eine globale ID ist eindeutig einem Modul zugewiesen, kann aber verwendet werden, um eine Ressource direkt zu referenzieren - beispielsweise kann die Globale ID 212 mit `http://localhost:8080/@212` aufgerufen werden, mithilfe eines anonymen Links (siehe Nutzermodul) auch durch Gäste.

4.1.3 Go-API

Die Go-API ist in GoDoc dokumentiert und ist unter https://godoc.org/git.mo-mar.de/masp/masp zu finden.

4.2 Nutzermodul

4.2.1 Funktionsumfang

Das Nutzermodul ist für die Verwaltung von Benutzerkonten und für die Überprüfung von Berechtigungen zuständig.

4.2.2 Berechtigungen prüfen

```
import (
    "git.mo-mar.de/masp/users"
    "github.com/gin-gonic/gin"
)
func handler(c *gin.Context) {
    // Wenn eine Ressource per ID aufgerufen wird, kann mit HasLinkAccess
  überprüft werden, ob Gast-Lesezugriff auf die Ressource besteht. Existiert
  keine ID, sollte `m.GetPrefixByModule(module)` als ID verwendet werden.

    id, _ := strconv.Atoi(c.Param("id"))
    if !users.HasLinkAccess(c, id) {
        // RequireUser prüft auf angemeldete Nutzer, RequireAdmin auf
  Projektadministratoren
        if !users.RequireUser(c) {
            // RequireUser kümmert sich automatisch um alle Fehlermeldungen!
            return
        }
    }
    // ...
    if users.IsAdmin(users.GetUserIDFromContext(c)) {
        // User ist Admin (ohne Fehlermeldung überprüfen)
        // ...
    }
    if users.GetUserIDFromContext(c) > 0 {
        // User ist angemeldet (ohne Fehlermeldung überprüfen)
        // ...
    }
}
```

4.2.3 Anonyme Links

Mit `users.GenerateLink(nil, id)` kann ein anonymer Link für Lesezugriff generiert werden - per Konvention sollte dieser bei jeder Ressource im `Masp-Permanent-Link` Header mitgesendet werden.

4.2.4 Go-API

Die Go-API ist in GoDoc dokumentiert und ist unter https://godoc.org/git.mo-mar.de/masp/users zu finden.

4.3 Aufgabenmodul

4.3.1 Funktionsumfang

Das Aufgabenmodul ist für die Verwaltung von Aufgaben verantwortlich und liefert dafür im Komplettpaket Code für Frontend, Backend und Datenbank.

4.3.2 Go-API

Die Go-API ist in GoDoc dokumentiert und ist unter https://godoc.org/git.mo-mar.de/masp/tasks zu finden.