Posty - Project 1

Project

Group: Group01

Projectname: Posty

Groupmembers: Benedikt Lang (mail at blang.io)

Hosting

Hosted on AWS Elastic Bean Stalk: https://posty.elasticbeanstalk.com/

Root CA: `./rootCA.pem`

Software, Tools, Libs

Golang

The original suggestion was to use `java` but Prof. Reiser agreed on an approach in `Go`. Golang was developed by google engineers to build large long-running systems and is especially suited for webservices.

The benefits:

- Most code is more concise than in `java`
- Lowlevel oauth2/oidc implementation more easy to implement (see later)
- Single binary deployment, Fast compilation, Dependency management
- Clear Codestyle and Docs

WGO (Go Workspaces)

wgo is a small wrapper around the `go` toolchain to enable a standalone workspace with fixed dependencies. Therefore `posty` is build using `wgo build posty` instead of `go build posty`.

Docker

The `EBS` Deployment uses docker, because it fits the picture and give greater flexibility. The project is build to one static binary (posty backend) and the frontend directory (html, javascript,..).

Since `EBS` supports native golang deployments, this would be possible too but it's more complicated to build the binary using the fixed dependencies bundled by wgo.

For convenience everthing can be build using docker, no need to install a `node.js` or `golang` environment (see later).

AngularJS, Bootstrap (Frontend)

A proper REST API needs a suitable consumer. AngularJS is a good choice to build an interactive frontend consuming a REST API. The whole application logic is in `frontend/app/static/scripts/controllers/main.js`. The view in `views/`.

Bootstrap is used to please the eye at least a little bit.

Important libraries (Backend)

- xhandler: HTTP Handler wrapper using net/context. Better flexibility between different routers, uses `context` to transport `sessions` etc.
- goji: Minimalistic Webframework. Gojis http router was used for speed and support for url patterns like `/api/post/:id`.
- gorilla/sessions/gorilla/securecookie: Secure Cookies (transporting the `user_id`) use HMAC and Encryption to be tamper-proof. It enables to scale the application horizontally since no session store is needed.
- dgrijalva/jwt: JWT parses and verifies id tokens as JSON Webtokens using available signing methods like HMAC or RSA.

Design

Workflow

- User visits `/` (gets redirected to `/login`)
- User chooses identify provider (Google or Paypal) and gets redirected to '/logingoogle`or` /loginpaypal`)
- Backend generates random state and nonce used for oidc. Redirects to provider with oidc clientid, state, scope...
- User gets redirected to provider
- User logs in on provider
- User gets redirected to callback endpoint: `/gcallback` (google) or `/pcallback` (paypal)
- Backend handles session creation, user is created if not already in database.
- User is redirected to board page `/`
- Posts are listed on page: Frontend calls `GET /api/posts` (with session cookie)
- Backend authenticates user based on session cookie (on every `/api/` call and returns result set from database.
- User posts something: Frontend handles REST Call: `POST /api/posts {"data":{"message":"my posting"}}`
- Backend responds with `201 Created` and responds with created post.
- User deletes post: Frontend handles REST Call: `DELETE /api/posts/423e7b0a-efcd-4eb4-9704-791f681507fa`
- If User is not authorized, API responded with Status 401, Error message is shown
- Backend responds with `204 No content` on success.
- User logs out: `/logout`. Session is invalidated by backend.

- User gets redirected to `/login`

Model (posty/model, posty/model/awsdynamo)

The model encapsulates the data store logic of the application. It's divided in two packages `user` and `post`, since those are the stored entities.

While the package `model` implements the interfaces and basic types, the package `awsdynamo` is the concrete implementation backed by AWS DynamoDB including integration tests.

OIDC (posty/oidc)

Google and Paypal were chosen as Identity Providers because both implement the OpenID Connect protocol (at least partly).

Because of problems with the Paypal API a general oidc library like coreos/go-oidc were not chosen in the final implementation: Paypals ID Token use a HMAC signature which can not be verified, because it's not signed with the users secret key. Paypal also does not support a proper "sub" Claim to identify the user without accessing to an userinfo endpoint.

The package oidc implements a suitable oidc strategy for Google and Paypal and was completely build by hand. `jwt-go` provides the necessary functionality to parse and verify the `id_token`.

Session stored secrets are used to verify the `state` of the oidc session and after the id token is verified, also the nounce and the audience is checked for any tampering attempt. To verify Googles id token googles certificates are loaded from the cert endpoint.

The paypal oidc is configured to work on the paypal sandbox for demostration purpose, this ensures better options for the demo (multiple accounts). Replacing the 3 oidc urls inside `oidc/paypal.go` from `sandbox.paypal.com` to `api.paypal.com` will work in production. No further changes are needed.

Controllers

The `controller` package consists of two important types `PostController` and `AuthController`. `AuthController` handles the process of logging in using OIDC and the registration of new users.

The `PostController` provides a REST API to create, delete and list posts.

Both controllers are connected with the model using flexible interfaces.

Middleware

Middleware is called before the actual http handler and provides necessary context. The `auth` middleware has two variants:

The `UnauthenticatedFilter` only allows logged-out users to reach the http handler, logged-in users are redirected. `AuthenticatedFilter` provides the exact opposite.

Both filters provide a flexible handling of the routes.

The `session` middleware handles the cookie managment. All session data is stored in cookies encrypted and hashed using the `securecookie` library. Using this approach, it's possible to scale horizontally without the need for a seperate session database.

Main

The main package builds the foundation for the project. It's build as a 12 Factor Application, which means it's completely configured using commandline flags and/or environment variables. This has a big advantage in deployment, since a seperate configuration can be supplied for development and production using the environment, like in Amazon Elastic Beanstalk.

Documentation

The code docs can be viewed using `godoc` in your webbrowser and is highly recommended (see below).

Frontend

The frontend is a very simple angularjs application.

It provides REST API calls and renders the result.

For user convenience it is possible to filter all posts using the filter field. The posts are filtered (all fields) as the user types.

Build, Test and Run

Build using Docker (recommended)

Requirements:

- Docker 1.7+

View Godocs

```
# Pull docker container
docker pull blang/posty-build-backend
# Exec godocs (127.0.0.1:6060)
./build/backend/godocs.sh
```

See http://127.0.0.1:6060/pkg/posty/ for GoDocs.

Build backend

```
# Pull docker container
```

```
docker pull blang/posty-build-backend
# OR build container locally
./build/backend/buildcontainer.sh

# Build ./posty using container
./build/backend/build.sh
```

Output: `./posty` binary

Build frontend

```
# Pull docker container
docker pull blang/posty-build-frontend
# OR build container locally
./build/frontend/buildcontainer.sh

# Build ./frontend/dist directory using container
./build/frontend/build.sh
```

Output: `./frontend/dist`

Build staging

After you built the backend and the frontend, the following structure should be present:

- ./posty (go binary)
- ./frontend/dist (frontend files)

Now you can build a production ready container:

```
./build/staging/buildcontainer.sh
```

This will create the container: `blang/posty-staging`

Build Elastic Beanstalk zip

You can create a valid zip file for Elastic bean stalk:

```
./build/staging/create_ebs_zip.sh
```

This will create `posty-staging.zip`.

Build manually

Frontend and backend can also be build without using docker:

View Godocs

Requirements:

- Go 1.4+

```
## Get wgo
go get github.com/skelterjohn/wgo

## Restore dependencies
wgo restore

GOPATH=$(wgo env GOPATH) godoc -http ':6060'
```

See http://127.0.0.1:6060/pkg/posty/ for GoDocs.

Build backend

Requirements:

- Go 1.4+

Build using `wgo` (for vendoring):

```
## Get wgo
go get github.com/skelterjohn/wgo

## Restore dependencies
wgo restore

## Build
wgo build posty
```

Build frontend

Requirements:

- Nodejs

```
cd frontend

npm install -g grunt-cli bower

## Install local dependencies
npm install

## Bower install
bower install

## Build
grunt build
```

Output: `./dist`

Run locally

Requirements:

- dynamodb local running on http://127.0.0.1:8000/
- local golang dev environment
- built `./posty`

Setup a development environment for the 12-factor app:

```
export POSTY_OIDC_GOOGLE_CLIENT_ID=[YOUR DATA HERE]
export POSTY_OIDC_GOOGLE_CLIENT_SECRET=[YOUR DATA HERE]
export POSTY_OIDC_PAYPAL_CLIENT_ID=[YOUR DATA HERE]
export POSTY_OIDC_PAYPAL_CLIENT_SECRET=[YOUR DATA HERE]
export POSTY_DYNAMODB_ENDPOINT=http://127.0.0.1:8000/
## Setup aws credentials or use ~/.aws credentials
export AWS_REGION=us-west-2
export AWS_ENDPOINT=http://127.0.0.1:8000/
export AWS_PROFILE=dev
export AWS_ACCESS_KEY_ID=dev
export AWS_SECRET_ACCESS_KEY=dev
```

Run the integration tests. This will create the dynamodb tables `user` and `post` needed.

```
wgo test posty/model/awsdynamo/integrationtest -test.v -integration
```

Run Posty:

```
./posty -frontend-path "./frontend/dist"
```

All configuration is done by commandline flags or environment variables beginning with `POSTY_`. Have a look at `./posty --help` for more information.