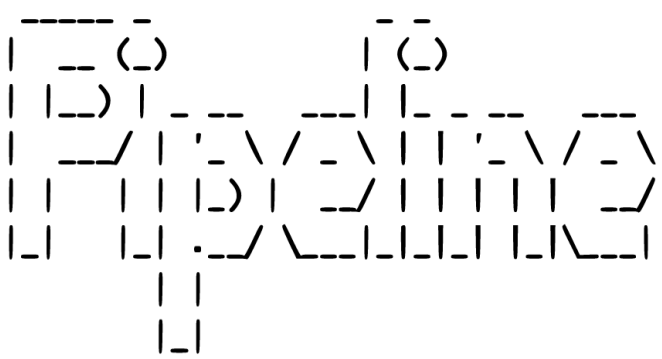


Et web-rammeverk for Python

Skrevet av

Ole-Jørgen Andersen

Studentnummer: 202154

Innhold

Innhold **1**

Introduksjon **2**
Beskrivelse av rammeverket 2

Bakgrunn **3**
Inspirasjon fra andre rammeverk 3

Metode **4**
Steg 1: API design spesifikasjon 4
Steg 2: API design 4
Steg 3: Implementasjon 4
Steg 4: Brukertesting 5

Prosess **6**
API design spesifikasjon 6
Scenario 1 6
Scenario 2 7
Scenario 3 8
Scenario 4 9
Scenario 5 10
Scenario 6 10
Scenario 7 11
Scenario 8 12
API design 13
Designprinsipper 13
Design mønstre 13
Personlige preferanser 15
Implementasjon 16
Brukertesting 17
Iterasjon 1 17
Iterasjon 2 20

Resultat **22**
Overordnet 22
Løsning på scenarioene 23

Diskusjon **26**

Introduksjon

Denne dokumentasjonen tar for seg rammeverket *“Pipeline”* og designprosessen som fant sted under utviklingen av rammeverket.

Utviklingen av rammeverket har ikke vært et gruppeprosjekt og er utviklet av en person, med unntak av brukertesting hvor flere parter har vært involvert, både grupper og enkeltpersoner.

Beskrivelse av rammeverket

Pipeline er et web-rammeverk for å bygge webapplikasjoner. Rammeverket skal gi brukeren et lavterskel verktøy å lage skreddersydde webapplikasjoner med, samtidig som det er åpent for at brukeren kan utvide funksjonaliteten.

Rammeverket har ikke som mål å inkluderer avansert funksjonalitet, som f.eks. skreddersydd autentisering, men inkluderer det essensielle, samt en rekke metoder for å lette vanlige og repetitive operasjoner som webapplikasjoner ofte byr på. Dessuten, så gir rammeverket brukeren mulighet til utvide APIet med egen funksjonalitet. Så hvis ønskelig, kan brukeren utvikle egne autentiseringsmekanismer.

I korte trekk så er målet med rammeverket at det skal være lett å ta i bruk, lett å bygge i og lett å vokse i.

Bakgrunn

Det finnes en rekke konkurrenter innen dette feltet med langt mer fasjonabel funksjonalitet. Eksempler på slike rammeverk kan være *“Django”* og *“Fast API”*, som begge er elsket høyt, da de leverer et godt API med bred funksjonalitet. Pipeline inkluderer bare en håndfull av denne funksjonaliteten, men er bygd på en lagdelt arkitektur, slik at man lett kan legge til mer funksjonalitet.

Pipeline bygger på et lavnivå bibliotek kalt *“WSGI”*. Dette er et godt bibliotek og man kan fint bygge webapplikasjoner med det, men det mangler flere praktiske abstraksjoner, noe som gjør utviklingsprosessen svært tungvint. Pipeline løser dette ved å bygge et abstraksjonslag rundt WSGI. Rammeverket gir brukeren et enkelt og uavhengig API lag, uten implementasjons detaljene til WSGI. Ytterligere, tilbyr rammeverket en rekke høyere abstraksjoner for å lette vanlige oppgaver, som f.eks. å returnere json data og sette cookies.

Inspirasjon fra andre rammeverk

Den endelige versjonen av rammeverket har i hovedsak tatt inspirasjon fra Fast API – et rammeverk som også er bygd på WSGI plattformen. Det kanskje mest fremtredende er et liknende API for å spesifisere endepunkter. I utgangspunktet ble det valgt et annet mønster for denne delen av APIet, men gjennom brukertesting kom det fram at Fast API sin løsning var å foretrekke. Det ble derfor besluttet å implementere en variant av denne løsningen, hvor enkelte deler ble endret. Nedenfor kan du se to klientkode eksempler, som løser det samme scenarioet. Ett fra Fast API og ett fra Pipeline.

```
# Fast API
@app.get('/user/{id}')
def index_view(id):
    return id

# Pipeline
@app.get('/user/(id)')
def index_view(req):
    return Response(req.params.id)
```

Metode

Følgende er metodikken bak stegene i designprosessen til rammeverket. Designprosessen bygger på Scenario-drevet design, noe som gjør at Pipeline er utviklet med fokus på hvilke oppgaver brukeren skal løse. Designprosessen har en iterativ del (steg 2-4), hvor rammeverket forbedres for hver iterasjon.

Steg 1: API design spesifikasjon

Dette er idemyldring steget. Målet er å komme opp med forslag til kode for å løse et gitt problem som rammeverket ditt vil møte. Denne koden er kalt *"klientkode"* og er et forslag til hvordan APIet ditt kan se ut. Ideelt sett bør det skrives flere forslag til klientkode per scenario, slik at man kan utforske ulike måter å løse problemet på. Ved å gjøre dette, kan vi finne bedre måter å løse scenarioet vårt på.

Overordnet har prosessen vært som følger:

1. Lag et relevant scenario som rammeverket skal løse
2. Lag ulike forslag til klientkode for å løse scenarioet

Steg 2: API design

Dette er det første steget i den iterative delen av prosessen. Her fokuserer vi til å begynne med på klient koden fra spesifikasjonen. Vi drøfter rundt hva vi burde bruke i en første iterasjon av APIet, på bakgrunn av designprinsipper, design mønstre, og personlige preferanser. Dette danner grunnlaget for hvordan selve APIet ser ut i form av f.eks. klasser og metoder. Deretter gjentas dette steget i en iterativ prosess, hvor vi går vekk fra å se på spesifikasjonen og eller fokuserer på å forbedre APIet basert på implementasjon og brukertesting.

Steg 3: Implementasjon

Dette er det andre steget i den iterative delen av prosessen. Fokuset ligger nå på å implementere logikken bak det APIet vi designet i forrige steg. Her er det viktig å abstrahere bort eksterne biblioteker og implementasjons detaljer, slik at det ikke lekker ut i vårt eget API. Ved å gjøre det, sørger vi for at APIet er fri for avhengigheter, da det kan skape inkompatibiliteter i senere iterasjoner.

Steg 4: Brukertesting

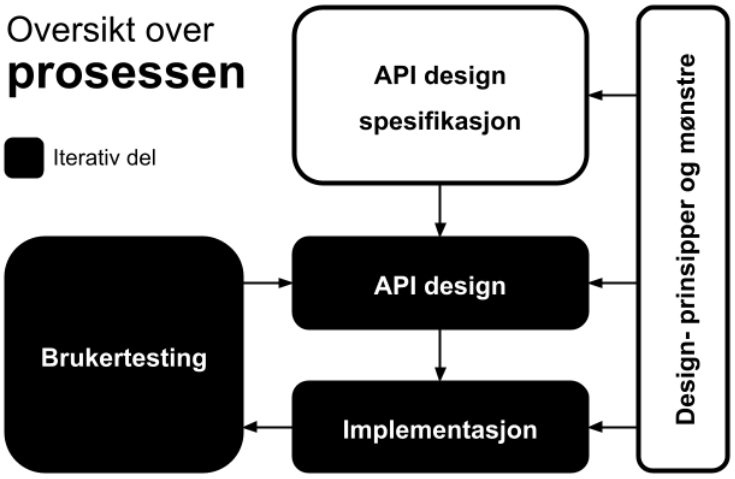
Dette er det siste og viktigste steget i den iterative delen av prosessen. Brukertesting gir oss tilbakemelding på hva vi har produsert i de tidligere stegene, fra målgruppen til rammeverket. Dette kan være både positive og negative momenter. Disse danner grunnlaget for forbedringer til APIet og/eller implementasjoner. Tilbakemelding er viktig, da det ofte er lett å tro at eget design er det beste, selv om det ikke alltid er det. Flere iterasjoner med brukertesting kan produsere viktige tilbakemeldinger, som utvilsomt vil lede til et bedre rammeverk.

Overordnet har brukertestingen foregått som følger:

1. Rammeverket blir sendt til en test kandidat, med følgende innhold:
    a. Rammeverket i form av en python pakke
    b. API dokumentasjon
    c. Scenarioer
2. Kandidaten blir bedt om å bruke rammeverket for å løse scenarioene, ved hjelp av API dokumentasjonen
    a. Hvis kandidaten sitter fast gis det tips fra utvikler
3. Undersøker og vurderer endringer på API og/eller implementasjoner, basert på tilbakemeldingene fra kandidaten

Prosess

Dette kapittelet tar for seg utviklingsprosessen til rammeverket; API design spesifikasjonen, API design, implementasjon, brukertesting og videreutvikling.



API design spesifikasjon

Scenario 1

Lag en webapplikasjon på port 3000 med et endepunkt “/”, som returnerer teksten: “Hello World!”.

```
# Forslag 1

from pipeline import Pipeline

app = Pipeline()

def index_view():
    return "Hello World!"

app.get(r'/', index_view)

app.run(3000)
```

```python
# Forslag 2

from pipeline import Pipeline

app = Pipeline(port=3000)

@app.endpoint('/')
def index_view(req):
    if (req.method == 'GET'):
        return "Hello World!"

app.run()
```

Scenario 2

Lag et endepunkt “/users” som returnerer en liste av brukere, i form av et array av json objekter.

```python
# Forslag 1

users = [
    {'name': 'Adrian'},
    {'name': 'Sebastián'}
]

def users_view():
    return JSONResponse(users)

app.get(r'/users', users_view)
```

```python
# Forslag 2

users = [
    {'name': 'Adrian'},
    {'name': 'Sebastián'}
]

@app.endpoint('/users')
def users_view(req):
    if (req.method == 'GET'):
        return users
```

Scenario 3

Lag et endepunkt “/user” med metode “post”, som tar imot en bruker og legger den inn i en liste av brukere. Når brukeren er lagt til, skal endepunktet returnere teksten “User created!” og statuskoden 201. Brukeren er et json objekt.

```
# Forslag 1

users = [
    {'name': 'Adrian'},
    {'name': 'Sebastián'}
]

def create_user(req):
    user = JSONParser(req.body)
    users.append(user)
    return Response('User created!', 201)

app.post(r'/users', create_user)
```

```
# Forslag 2

users = [
    {'name': 'Adrian'},
    {'name': 'Sebastián'}
]

@app.endpoint('/users')
def create_user(req):
    if (req.method == 'POST'):
        users.append(req.body)
        return ('User created!', 201)
```

Scenario 4

Lag et dynamisk endepunkt “/users/(index)” som tar imot en variabel “index” fra nettadressen, og returnerer en bruker fra en liste av brukere med korresponderende index. Brukeren er et json objekt.

```
# Forslag 1

users = [
    {'name': 'Adrian'},
    {'name': 'Sebastián'}
]

def user_view(req):
    index = req.params[0]
    return JSONResponse(user[index])

app.get(r'/users/(.*)$', user_view)
```

```
# Forslag 2

users = [
    {'name': 'Adrian'},
    {'name': 'Sebastián'}
]

@app.endpoint('/users/(index)')
def user_view(req):
    if (req.method == 'GET'):
        return users[req.params.index]
```

Scenario 5

Lag et endepunkt “/search” som henter ut query parameteret “q” fra nettadressen (?q=), og returnerer teksten: “You searched for (q)”.

```
# Forslag 1

def search_view(req):
    return "You searched for " + req.query['q']

app.get(r'/search', search_view)
```

```
# Forslag 2

@app.endpoint('/search')
def search_view(req):
    if (req.method == 'GET'):
        return "You searched for " + req.query.q
```

Scenario 6

Lag to endepunkter “/pay” og “/receipt”, hvor det først endepunktet omdirigerer brukeren til det andre. Endepunkt “/receipt” skal returnere teksten: “Here is your receipt!”.

```
# Forslag 1

def pay():
    res = Response()
    res.redirect(r'/receipt'')
    return res

def receipt():
    return "Here is your receipt!"

app.get(r'/pay', pay)
app.get(r'/receipt', receipt)
```

```
# Forslag 2

@app.endpoint(r'/pay')
def pay(req):
    if (req.method == 'GET'):
        return endpoint(r'/receipt')

@app.endpoint(r'/receipt')
def receipt(req):
    if (req.method == 'GET'):
        return "Here is your receipt!"
```

Scenario 7

Lag et endepunkt “/product” som lagrer en cookie hos klienten, med navn “product” og verdien “visited”.

```
# Forslag 1

def product_view():
    res = Response()
    res.set_cookie('product', 'visited)
    return res

app.get(r'/product', product_view)
```

```
# Forslag 2

@app.endpoint(r'/product')
def product_view(req):
    if (req.method == 'GET'):
        return ('', 200, Cookie('product', 'visited))
```

Scenario 8

Lag et endepunkt “/greeting” som returnerer en html fil “greeting.html”, med innhold: “`<h1>Hello {name}!</h1>`”, hvor “name” er en injisert variabel som skal settes til “Armin”.

```
# Forslag 1

# Fil: greeting.html

<h1>Hello {name}!</h1>

# Fil: app.py

def greeting_view():
    vars = {'name': 'Armin'}
    return TemplateResponse('greeting.html', vars)

app.get(r'/greeting', greeting_view)
```

```
# Forslag 2

# Fil: greeting.html

<h1>Hello ((name))!</h1>

# Fil: app.py

@app.endpoint(r'/greeting')
def greeting_view(req):
    if (req.method == 'GET'):
        res = Template('greeting.html')
        res.vars['name'] = "Armin"
        return res
```

API design

Dette er tankene rundt valgene tatt for det endelige APIet, på bakgrunn av designprinsipper og mønstre, samt personlige preferanser.

Designprinsipper

Low barrier to entry

Som nevnt i innledningsvis, er målet med rammeverket at det skal være lett å ta i bruk, lett å bygge i og lett å vokse i. Første steg mot dette målet, er å gi brukeren en såkalt “low barrier to entry”. Altså at rammeverket er lett å ta i bruk. For å oppnå dette, er APIet designet slik at det er lite “boilerplate” kode som må skrives for å sette opp en web applikasjon. Dette er oppnådd ved å tilby standardverdier på parametere, som brukeren kan overskrive om ønskelig.

Self-documenting code og Least astonishment

For å gi brukeren et rammeverk som er lett å bygge i, har APIet blitt designet med tanke på “self-documenting code”. Det vil si at APIet benytter “menneskelig” lesbare navn på variabler, metoder, klasser og parametere. Slik blir det er letter å forstå hva den delen av APIet gjør, og brukeren blir ikke overrasket over resultatet. En positiv bieffekt ved dette, er at det gjør rammeverket lettere å ta i bruk, ved at APIet blir enklere å forstå ved første glans.

Layered architecture

I tillegg, er APIet lagdelt. Noe som har gjort det mulig å konstruerer abstraksjoner for å lette vanlige og repetitive operasjoner som webapplikasjoner ofte byr på, slik som å sende json data til klienten istedenfor bare alminnelig tekst. Det åpner også opp for at brukeren kan utvide funksjonaliteten selv i gjennom arv. I det store bilde, bidrar det til å gjøre lette ting lette å gjøre og vanskelige ting mulige å gjøre.

Design mønstre

Da python skiller seg noe fra de objektorienterte språkene som disse designmønstrene var originalt laget i, vil omfanget brukt i APIet til tider skille seg litt ut. APIet implementere derfor hovedprinsippene fra disse mønstrene, og ikke nødvendigvis alltid foreslått syntax.

Det har også vært fokus på å bruke riktig mønster på rett plass, da unødvendig og feil bruk, kan resultere i et tungvint API. Python inkluderer sine egne løsninger på problemer som mønstre ofte var designet for å løse. Disse vil prioriteres framfor mønstre, da tvangs implementering av mønstre kan lede til unødvendig boilerplate kode.

Request-response pattern

Først og fremst, er selve rammeverket bygd på "Request-response" mønsteret. Dette kommer til syne i APIet ved at kontroll funksjonene tar i mot et "Request" objekt og returnerer et "Response" objekt. Hvilket er basert på klientkode eksemplet fra forslag 1 i scenario 3. Dette er brukt da det følger logikken bak "http" kommunikasjon, noe som mange kjenner til. En fordel ved å bruke et slikt kjent mønster, er at det vil være enklere å forstå flyten i APIet, og derfor bidra til å gjøre rammeverket enklere å ta i bruk.

En annen fordel med dette mønsteret, er at det gir oss en naturlig fasade å gjemme implementasjons detaljer bak, samtidig som det lar oss eksponere abstraksjoner for å legge til og hente ut informasjon, som f.eks. statuskode og cookies.

Builder pattern

I et forsøk på å gjøre det enklere å bygge webapplikasjoner, ble "builder" mønsteret brukt i den overordnede prosessen av å opprette en web applikasjon. Istedenfor å ta alle parametere med en gang, tar vi ett parameter av gangen ved hjelp av metoder. Når applikasjonen er klar, kaller vi en metode for å sette alt sammen og kjøre applikasjonen. Dette skal gjøre det mulig å lage store applikasjoner på en oversiktlig måte. Denne måten å strukturere byggeprosessen på, er en rød tråd i alle klientkode forslagene fra scenarioene.

Det ble også forsøkt å bruke "builder" mønsteret i andre komponenter, som f.eks. i request og response, men dette resulterte i unødvendig kode og gjorde APIet tungvint, da Python har mulighet for å bruke navngitte parametere i konstruktører. Siden disse komponentene ikke krevde mange parametere, ble dette feil mønster for situasjonen.

Strategy pattern og dependency injection

For å gjøre det enkelt å bygge i rammeverket, ble "strategy" mønsteret. Prinsippet bak mønsteret kan ses i hvordan endepunktene fungerer. Hver rute har en ulik strategi i form av en kontroller funksjon, som blir injisert som en avhengighet til endepunktet. Denne definerer oppførselen i det gitte endepunktet.

I det endelige APIet ble det tatt utgangspunkt i forslag 1 fra scenario 1 for å navngi metodene for endepunkt. En grunn til det var at forslaget brukte navnet på metodene fra http protokollen, altså "get", "post", osv... Ideen bak dette er at APIet blir enklere, i motsetning til forslag 2, hvor det er langt mer boilerplate kode. For i forslag 2 må man sjekke hvilken metode som ble kalt i kontroll funksjonen, mens i forslag 1 er det gjemt bort i implementasjonen.

Decorator pattern

Under brukertesting, ble det besluttet å endre den delen av APIet ansvarlig for å binde ruter til kontrollere, til å bruke "decorator" mønsteret. Hovedargumentasjonen for dette var at det fort kunne bli uoversiktlig kode, hvis man hadde mange endepunkter. Mønsteret løser dette ved samle kallet til rute metoden sammen med definisjonen av kontrolleren.

Iterasjon 1 i brukertesting seksjonen, forklarer hvordan deler av forslag 1 og 2 i scenario 1 ble kombinert for å lage et nytt API, basert på decorator mønsteret.

Personlige preferanser

Type hints, docstrings og API dokumentasjon

En utfordring med å lage et rammeverk i Python er at det ikke er et statisk-type språk. Grunnen til dette er at brukeren ikke nødvendigvis får hint til datatyper i IDEet sitt, slik som du får i f.eks. Java. Problemet med dette er at det ikke alltid er opplagt hva slags datatyper som støttes, noe som kan gjøre APIet tungvint. For å løse dette, har det vært fokus på å aktivt legge til såkalte "type hints" i APIet. Dette er en relativt ny funksjon i Python og lar oss gi hint til brukeren om hvilke datatyper man burde bruke, liknende hva man finner i Java.

Likevel kan det oppstå uklarheter. Derfor har det også vært fokus på å dokumentere hva variabler, metoder, klasser og parametere i APIet gjør, ved hjelp av "docstrings".

Disse finnes som “hints” i IDEet. For å gi brukeren en oversikt over hele APIet, har det blitt utviklet en API dokumentasjon.

MVC og CRUD ordlegging

Selv om det ikke nødvendigvis er relevant for selve rammeverket, er det verdt å nevne at det har vært bevisst bruk av terminologi som “MVC” (Model View Controller pattern) og “CRUD” (Create Read Update Delete) i docstrings, parametere og eksempelkode vist i API dokumentasjonen. Eksempler på dette er bruk av frasen “kontroller funksjon”, samt funksjonsnavn som “create_users” og “users_view”. Grunnen til dette er at slik navngiving ofte er brukt i lignende rammeverk, noe som kan gjøre overgangen lettere for enkelte brukere.

DRY prinsippet

For å holde kildekoden til rammeverket kompakt, har det vært fokus på å dele opp APIet opp i komponenter. Disse komponentene kan gjenbrukes og utvides ved arv. Dette står i stil med prinsippet om “DRY” (Don’t Repeat Yourself) kode. Målet her er å gjenbruke mest mulig, hvilket gjør vedlikehold av rammeverket lettere og anses derfor også som god praksis. Dette kan ses i forslag 1 i en rekke scenarioer, hvor abstraksjonene for request og response er gjenbrukt flere steder.

Implementasjon

Facade pattern

I det store bilde bygger hele APIet på “facade” mønsteret, ved at det er en høyere abstraksjon av WSGI. For å holde brukerens kode uavhengig av WSGI, var det viktig å skjule implementasjonen, slik at APIet til WSGI ikke lakk ut i rammeverkets eget API. Under implementasjon var det derfor fokus på å abstrahere bort WSGI, blant annet ved å konstruere metoder som gjorde kall til WSGI bak fasaden. Eksempler på dette er “set_cookie” og “redirect” metodene.

I tilfeller der klasse attributter involverte implementasjon mot WSGI, var Pythons egen “getter” og “setter” funksjonalitet å foretrekke. I motsetning til vanlige metoder, lar denne innebygde funksjonaliteten oss bruke attributter på ordinær måte, istedenfor å bruk av metodekall. Tanken bak dette er at det gir brukeren et mer konsistent API, hvor alle attributter anvendes på samme måte. Håpet er at dette bidrar med å gi brukeren en økt “low barrier to entry”.

Ved å ta i bruk fasade mønsteret og abstrahere bort WSGI, var målet å skape et uavhengig API lag. Ved å legge vekt på en slik lagdelt arkitektur, kunne vi skape et rammeverk som holdt brukerens kode adskilt fra implementasjonen, noe som vil gjøre det lettere å vedlikeholde og bytte ut implementasjonen i framtiden.

Feilhåndtering

Med tanke på low barrier to entry og self documenting code, er det også viktig med god feilhåndtering og kommuniserende feilmeldinger. Følgelig har det vært fokus på å fange errorer der de kan oppstå og gi en beskrivende feilmelding til brukeren. Dette har i hovedsak blitt implementert med forebyggende feilhåndtering, som f.eks. sjekking av syntax for endepunkt ruter, men også ved hjelp av “try-catch” metodikk.

Singleton pattern

Utover dette, har ideen bak “singleton” mønsteret blitt implementert i metoden som starter applikasjonen. Grunnen til dette er at det kun kan være en kjørende web applikasjon per port. Hvis det er flere kjørende web applikasjoner på samme port, kan det oppstå uforventet og uønsket oppførsel. Dette bidrar og til bedre feilhåndtering, ved å nekte bruker å starte enda en applikasjon på samme port.

Brukertesting

Iterasjon 1

Dette var en tidlig versjon av rammeverket. APIet var tilgjengelig for testerne, men uten noen implementasjoner. Dette var forsåvidt greit, da den første iterasjonen fokuserte på å teste selve APIet. Testerne fikk tilsendt rammeverket sammen med API dokumentasjon og scenarioene. De ble så bedt om å skrive kode for å løse scenarioene med hjelp fra API dokumentasjonen.

Men, som en bonus for første iterasjon, ble testerne bedt om å forsøke å løs scenarioene uten hjelp fra API dokumentasjonen først. Hensikten med dette var å se hvor lav eller høy terskelen for å komme i gang var.

Tilbakemeldinger

1. Noe vanskelig å komme i gang uten eksempel kode
2. Defineringen av endepunkter blir fort uoversiktlig når det er mange
3. Tungvint API for å spesifisere ruter, da det bruker regex uttrykk

Endringer

- Vanskeligheter med å komme i gang

Testerne satte seg fast allerede på første scenario. Dette var forsåvidt forståelig, da det var flere steder hvor type hints og docstrings ikke var implementert helt korrekt. Utvikler ga derfor tips, og feil implementasjonen skal rettes opp i til neste iterasjon. Til tross for dette, var det vanskelig å vite hvor man skulle starte, uten ihvertfall å ha sett et såkalt “quick start” eksempel. Dette er egentlig å forvente, og jeg vil ikke si at dette bidrar til noen særlig høy terskel for nykommere.

- Nytt endepunkt API

Det finnes mange måter strukturere endepunkter på, men for å unngå store endringer til APIet, så ble det besluttet å refaktorere koden til å bruke “decorator” mønsteret. Dette gjør at man kan samle rute kallet og definisjonen av kontrolleren, slik at man unngår spredt kode. Resultatet blir logisk seksjonert kode, som er mer oversiktlig.

For å finne et nytt endepunkt API, gikk jeg tilbake til scenario 1, hvor forslag 2 foreslo bruk av decorator mønsteret. Problemet var bare at klientkoden i forslag 2 inkluderte mer boilerplate kode, da man må sjekke hvilken http metode som er i bruk. Derfor ble det besluttet å ta endepunkt metoden fra forslag 1 og integrere den i forslag 2. Dette eliminerte behovet for boilerplate koden, og resulterte et bedre API.

```
# Gammelt endepunkt API

def index_view():
    pass

app.get(r'/, index_view)
```

```
# Nytt endepunkt API

@app.get(r'/')
def index_view(req):
    pass
```

- Nytt rute API

For å håndtere det tungvinte rute APIet, ble det besluttet å erstatte den regex baserte metodikken. Motivasjonen bak å bruke regex i utgangspunktet, var at implementasjonen ville bli enklere. Resultatet ble et mer konvensjonelt API, sett i andre liknende rammeverk. Dette er ikke så dumt, da det kan bidra til å gi enkelte brukere en lavere terskel for å komme i gang med rammeverket.

For å finne et nytt rute API, gikk jeg tilbake til scenario 4, hvor forslag 2 foreslår et alternativ til bruk av regex. Klientkoden til forslag 2 viser bruk av en nøkkel “index” i strengen. En positivt bieffekt av en slik vinkling er at vi kan hente ut navnet på nøkkelverdien og bruke den i APIet for å hente ut parametere senere. Hvilket gjør at vi kan fjerne det liste baserte APIet, noe som gir oss et mer selvforklarende API.

```
# Gammelt rute API

@app.get(r'/users/(.*)$')
def users_view(req):
    index = req.params[0]
```

```
# Nytt rute API

@app.get('/users/(index)')
def users_view(req):
    index = req.params['index']
```

Iterasjon 2

På dette punktet var APIet implementert og testerne kunne endelige se hva koden deres produserte. Implementasjonen fungerte for det meste, med unntak av noen uforutsette feil til å begynne med. Testerne hadde i tillegg noen ønsker. Testoppsettet var det samme som i iterasjon 1.

Tilbakemeldinger

1. Ønske om abstraksjoner for ofte brukte feilmeldinger
2. Ønske om kommuniserende feilmeldinger i implementasjonen, spesielt:
    a. Når en bruker har definert to like endepunkter
    b. Når det ikke finnes et endepunkt for en forespurt rute
    c. Når http metoden brukt, ikke støttes

Endringer

- Forbedringer i brukertest metodikk

Når rammeverket ble kjørt hos testerne, viste det seg raskt at implementasjonen hadde kritiske feil. Disse var relatert til manglende miljøvariabler på Windows OS, og raskt rettet opp i. En lærepenge å ta med seg fra dette, er at utvikler burde gjennomføre grundigere tester før brukertester, slik at testerne ikke stopper opp på grunn av grunnleggende feil i rammeverket.

- Nytt error API og bedre feilmeldinger

I arbeidet med å finne måter å forbedre feilhåndtering på, dukket det opp en mulighet for å kombinere testernes ønske om abstraksjoner for ofte brukte feilmeldinger, og ønske om kommuniserende feilmeldinger i implementasjonen, til en og samme løsning. Dette kunne gjøres ved å konstruere en responsklasse for vanlige feilmeldinger og kombinere den med en “exception” klasse for intern feilhåndtering. På den måten, kan man både bruke klassen i APIet for å returnere vanlige feilmeldinger, og internt i implementasjonen for å heve feil. Tanken bak dette er at det står i stil med DRY prinsippet, hvilket resulterer i færre linjer med repetitiv kode.

For å implementere dette, ble det først tenkt å bruke “multi-inheritance” for å slå sammen klassene til en. Problemet med det er at exception APIet ville ha blandet seg med response APIet, hvilket ville gitt brukeren et uoversiktlig API, noe som hverken står i stil med “self-documenting code” eller “low barrier to entry”. Grunnet

dette, endte den endelige implementasjonen opp med å bruke “composite pattern”. Istedenfor at begge klassene er i familie, så inkluderer exception klassen et response objekt, som inneholder en passende feilmelding.

Dette er ikke en perfekt løsning, da det gir et litt annerledes API fra det brukeren er vant med fra de andre responsklassene. Noe som høyner terskelen til nye brukere litt, men håpet er at god dokumentasjon kan gjøre opp for dette.

```
# Ny feilhåndtering i API

@app.get('/account')
def account_view(req):
    if not logged_in:
        return Error(401).response # body=401 Unauthorized
    else:
        return Response('Welcome to account!')
```

```
# Ny feilhåndtering i implementasjon

try:
    raise Error(500, 'No controller found for route /')
except Error as err:
    print(err)
    response = err.response # body=500 Internal Server E…
```

Resultat

Dette kapittelet presenterer det endelige rammeverket. Det tar for seg hva endelige rammeverket oppnår på bakgrunn av målene satt innledningsvis, og hvordan designprinsipper, designmønstre og brukertesting har formet APIet og implementasjonen bak.

Overordnet

Som nevnt tidligere, er målene for rammeverket at det skal være lett å ta i bruk, lett å bygge i og lett å vokse i. Jeg føler det endelige rammeverket etterlever disse målene, ved å gi brukeren et enkelt og kjent API, med den mest essensielle funksjonaliteten. I tillegg til en arkitektur som lar brukeren utvide rammeverket med ny funksjonalitet.

For å oppnå dette, har det vært viktig å sette noen overordnede design prinsipper for rammeverket fra starten av. Prinsipper som blant annet “Low Barrier to Entry”, “Self-documenting” og “Layered Architecture”, har påvirket valg av design mønstre for både API og implementasjonen bak. Det har derfor spilt en stor rolle i å forme det endelige rammeverket.

For å kvalitetssikre API og implementasjonen bak, har det vært kritisk å la eksterne parter få teste rammeverket. Faren ved å utvikle et rammeverk alene er at man fort kan ende opp med egosentriske løsninger, som bare er gode fra ditt eget perspektiv. Ved å involvere andre, har jeg fått gode tilbakemeldinger, som har dannet grunnlaget for flere revideringer av rammeverket. I disse revideringene har nye designmønstre blitt vurdert og implementert, men hele tiden med de overordnede designprinsippene og målene i tankene.

Løsning på scenarioene

Scenario 1

```
from pipeline import Pipeline

app = Pipeline()

@app.get('/')
def index_view():
    return Response('Hello World!')

app.run(3000)
```

Scenario 2

```
users = [
    {'name': 'Adrian'},
    {'name': 'Sebastián'}
]

@app.get(r'/users')
def users_view():
    return JSONResponse(users)
```

Scenario 3

```
users = [
    {'name': 'Adrian'},
    {'name': 'Sebastián'}
]

@app.post('/users')
def create_user(req):
    user = JSONParser(req.body)
    users.append(user)
    return Response('User created!', 201)
```

Scenario 4

```
users = [
    {'name': 'Adrian'},
    {'name': 'Sebastián'}
]

@app.get('/users/(index)')
def user_view(req):
    index = int(req.params[index])
    return JSONResponse(user[index])
```

Scenario 5

```
@app.get('/search')
def search_view(req):
    q = req.query['q']
    return Response(f'You searched for {q}')
```

Scenario 6

```
@app.get('/pay')
def pay_view():
    res = Response()
    res.redirect('/receipt'')
    return res

@app.get('/receipt')
def receipt_view():
    return Response('Here is your receipt!')
```

Scenario 7

```
@app.get('/product')
def product_view():
    res = Response()
    res.set_cookie('product', 'visited)
    return res
```

Scenario 8

```
# Fil: greeting.html

<h1>Hello {name}!</h1>

# Fil: app.py

@app.get('/greeting')
def greeting_view():
    vars = {'name': 'Armin'}
    return TemplateResponse('greeting.html', vars)
```

Diskusjon

Sett i det store bilde, vil jeg si prosjektet har gått veldig bra. Det har vært artig å få utvikle sitt eget rammeverk, og lærerikt å få gjøre det gjennom en scenario drevet prosess. Likevel har det vært en noe rotete opplevelse, da jeg selv ikke har holdt på med denne typen utvikling tidligere. Altså har det vært både positive og problematiske momenter under prosjektet. Disse ønsker jeg å diskutere i dette korte kapittelet.

Dårlig forhåndstesting

Et noe uventet problem som oppstod under brukertesting, var at jeg ikke hadde testet koden godt nok på forhånd. Slik at kritiske feil stoppet test kandidatene fra å utføre oppgavene de hadde fått. Dette skapte store forsinkelser i test iterasjonene. En lærepenge jeg har tatt fra dette, er at utvikler alltid burde sjekke om koden fungerer godt nok til at kandidaten ihvertfall kan starte på oppgavene sine. Det kan tenkes at en god måte å gjøre dette på, er å skrive enhetstester.

Antall test iterasjoner

En annen utfordring var å få inn flere test iterasjoner. Grunnen til dette var at jeg allerede hadde skrevet all implementasjons koden mellom iterasjon 1 og 2. Hvilket gjorde at det var lite å teste i en eventuell tredje iterasjon. Dessuten så hadde jeg et mangfold av testkandidater i iterasjon 1 og 2, og så derfor heller ikke noe behov for nye iterasjoner med flere kandidater.

Men i ettertanke, skal det sies at det kunne ha vært nyttig med en tredje iterasjon for å teste det nye error APIet, da det hadde et noe annet design i motsetning til resten av APIene. Ved å ikke teste dette, er det mulig at denne delen av APIet kan virke kronglete for folk flest å bruke.

For mye fokus på erfarne brukere

Et annet mulig problem som jeg har tenkt på i ettertid, er at fokuset på målgruppen for rammeverket kan ha vært noe feil. Grunnen til dette, er at under utvikling og revidering har inspirasjon til forbedringer ofte vært basert liknende rammeverk. I hovedsak Fast API.

Problemet ved dette, er at fokuset kanskje har vært mer på å etterligne, enn å lære av. Dette kan ha ført til at rammeverket er mer rettet mot erfarne brukere, istedenfor alle som trenger et web rammeverk, inklusiv de som ikke har benyttet liknende løsninger før. Løsningen kunne kanskje være brukertesting, men alle test kandidatene har vært erfarne web rammeverk brukere. Derfor kunne det i retrospekt, ha vært lurt å inkludere alle salgs testkandidater.

Erfaring

Utover punktene nevnt ovenfor, synes jeg prosjektet har gått veldig bra. Å utvikle basert på designprinsipper, mønstre og tilbakemeldinger har vært utfordrende, da det har vært mye nytt å sette seg inn i. Men på den andre siden, har det vært en lærerik prosess. Jeg sitter igjen med mye ny og god kunnskap, som vil gjøre utviklingen av lignende prosjekter i fremtiden enklere. Hvilket vil og bidra til bedre resultater.