# Transcripts - Hex Architecture

Also known as "Clean Architecture"

## Domain

Produce the text transcript of a customer-provided audio/video file.

## Submit request

Transcription requests are submitted using a REST API, POST /requests.

### Check status of a request

Customers check the status of their request using a REST API, GET /status/:id, since the transcription process can take seconds, hours or days depending on the customer-specified quality standard.

### Get transcript

Customers retrieve the text transcript using a REST API, GET /transcripts/:id.

## Model

The model describes the request object based on the requirements.

### Request Struct

Fields needed to fulfill the domain requirements.

package request

type Request struct {

RequestID string `json:"id" db:"id"`

CustomerID string `json:"customer" db:"cust\_id"`

MediaFileURI string `json:"mediafile" db:"mediafile"`

Status string `json:"status" db:"status"`

CreatedAt time.Time `json:"created" db:"created"`

CompletedAt time.Time `json:"completed" db:"completed"`

UpdatedAt time.Time `json:"updated" db:"updated"`

DeletedAt time.Time `json:"deleted" db:"deleted"`

}

* **RequestID** - uniquely identify each request so we can associate and report the status and resulting transcript of that request. Server generates.
* **CustomerID** - identify the customer making the request, so we can check the customer requesting the status of a request is the same customer that submitted the request. Client provides.
* **MediaFile** - the customer-provided file from which we produce the text transcript. Client provides.
* **Status** - the processing status of the request, e.g., "PENDING", "COMPLETED". Server generates.
* **CreatedAt** - date and time the request was accepted; i.e., not rejected for being obviously invalid. Server generates.
* **CompletedAt** - date and time the request was completed; i.e., final transcript available. Server generates.
* **UpdatedAt** - date and time the request was updated. Server generates.
* **DeletedAt** - date and time the request was (soft-)deleted. Server generates.

### Request Repository Interface

Abstract repository interface to allow needed operations.

package request

type RequestRepository interface {

Create(request \*Request) error

FindByID(id string) (\*Request, error)

FindByCustomer(customer string) ([]\*Request, error)

FindByStatus(customer, status string) ([]\*Request, error)

}

* **Create** - create a new request.
* **FindByID** - find one specific request.
* **FindByCustomer** - find all requests from one specific customer.
* **FindByCustomerStatus** - find all requests from one specific customer with the specified status.

## Service

Business logic implementation of the abstract repository RequestRepository above.

### Request Service

Most methods (other than CreateRequest) simply pass through to the underly repository implementation.

package request

type RequestService interface {

CreateRequest(request \*Request) error

FindRequestByID(id string) (\*Request, error)

FindRequestsByCustomer(customer string) ([]\*Request, error)

FindRequestsByStatus(customer, status string) ([]\*Request, error)

}

type requestService struct {

repo RequestRepository // implement the RequestRepository interface

}

func NewRequestService(repo RequestRepository) RequestService {

return &requestService{

repo,

}

}

func (s \*requestService) CreateRequest(request \*Request) error {

request.RequestID = uuid.New().String()

request.CreatedAt = time.Now().UTC()

request.Status = "PENDING"

return s.repo.Create(request)

}

func (s \*requestService) FindRequestByID(id string) (\*Request, error) {

return s.repo.FindByID(id)

}

etc...

* **RequestService** - interface implemented by this service.
* **requestService** - type (struct) that's the receiver for methods implementing the RequestService interface
* **NewRequestService** - return a new RequestService which wires in the underlying RequestRepository.
* **CreateRequest** - create a new request, applying business logic like assigning a unique RequestID.
* **FindRequestByID** - ask the repository for the specified request.

## Handler

*The handler allows us to push the information towards the center of the Hexagon, meaning that the inputs and outputs are solely based on the business logic. These layers of communication toward the domain are called “ports”. You can imagine that if someone wanted to create a listener on a queue or invoke a GRPC call the setups would be similar and you would expect the same response from the business layer. (*[*Go Hex Arch*](https://www.joeldholmes.com/post/go-hex-arch/)*)*

Functions as an adapter to the domain. Could go in package handler but often works better in the domain package, i.e., package request.

package request

type RequestHandler interface {

Create(w http.ResponseWriter, r \*http.Request, p httprouter.Params)

GetByID(w http.ResponseWriter, r \*http.Request, p httprouter.Params)

etc...

}

type requestHandler struct {

requestService RequestService // implement the RequestService interface

}

func NewRequestHandler(requestService RequestService) RequestHandler {

return &requestHandler{

requestService,

}

}

func (h \*requestHandler) GetByID(w http.ResponseWriter, r \*http.Request,

p httprouter.Params) {

id := p.ByName("RequestID")

request, \_ := h.requestService.FindRequestByID(id)

response, \_ := json.Marshal(request)

w.Header().Set(etc...)

w.WriteHeader(http.StatusOK)

\_, \_ = w.Write(response)

}

etc...

* **RequestHandler** - interface implemented by the request handlers.
* **requestHandler** - type (struct) that's the receiver for methods implementing the RequestHandler interface.
* **NewRequestHandler** - return a new RequestHandler which wires in the underlying RequestService.
* **GetByID** - ask the service for the specified request.

## Database Adapter

Persistent storage using PostgresSQL that fulfills the RequestRepository interface.

package psql

type requestRepository struct {

db \*sql.DB

}

func NewPostgresRequestRepository(db \*sql.DB) request.RequestRepository {

return &requestRepository{

db,

}

}

func (r \*requestRepository) Create(request \*request.Request) error {

r.db.QueryRow("INSERT INTO requests( ... ) " +

"VALUES ( $1 ... ) RETURNING id",

request.RequestID, ...).Scan(%request.ID)

return nil

}

etc ...

* **requestRepository** - type (struct) that's the receiver for methods implementing the RequestRepository interface.
* **NewPostgresRequestRepository** - return a new RequestRepository which wires in the underlying Postgres database connection.
* **Create** - add the request to the database.

## Main Application

Inject the repository into the service, the service into the handler, and serve the handler. Injection makes it easy to support additional databases later.

package main

func main() {

var repo = request.RequestRepository

pconn := postgresConnection("postgresql://postgres@localhost/request?sslmode=disable")

defer pconn.Close()

repo = psql.NewPostgresRequestRepository(pconn)

requestService := request.NewRequestService(repo)

requestHandler := request.NewRequestHandler(requestService)

[routing]

[ListenAndServe]

}

func postgresConnection(database string) \*sql.DB {

fmt.Println("Connecting to PostgreSQL DB")

db, err := sql.Open("postgres", database)

if err != nil {

log.Fatalf("%s", err)

panic(err)

}

return db

}