Overview

- Design considerations: pure code, error architecture.
- Demo following technologies:
  - JSON
  - Node Package Manager `npm`
  - Asynchronous programming using `async` and `await`.
  - Mongo db
  - Running external programs from nodejs.

Entities vs Values

- Entity objects have identity and changing state, value objects do not.
- Typically value objects are immutable.
- Examples:
  - An employee is an entity. Typically having multiple simultaneous instances of the same employee would be a problem.
  - Money would be a value.
  - A specific dollar bill would be an entity having a Money value.

Pure Functions

A **pure function** only computes a result which depends only on its parameters.

- It does not change any of its parameters.
- It does not have any **side-effects**: no changes to non-local state; **no I/O**.
- It cannot access any non-local variables.

Advantages of Pure Functions

- Code which uses pure functions are **referentially transparent**: i.e. the same expression always results in the same result.
- Easy to understand as each function can be understood in isolation.
- Possible to cache function results.
- Easy to test as it is not necessary to mock I/O like databases.

Disadvantages of Pure Functions

- A program without side-effects is pretty useless.
- Does not fit in with OO, which is built around objects having mutable state.
- Necessary to pass arguments via intermediate functions which have no interest in them.
- Cannot update data structures: only build new versions. No writable arrays; need to use **persistent data-structures**.

Pure Functions in the Real World

- Try to keep core of an application pure, keep impure code restricted to the edges of the application.
- Sometimes it is necessary to access global information like configuration information all over the program. Often dealt with by passing some kind of read-only context parameter between functions. However, can also use a non-local access as long as the access is read-only; think of it as an implicit parameter.
- Pure functional languages like Haskell support patterns to make this easier.

Error Architectures

- When code encounters an error, it outputs error message directly. **Totally unacceptable** as all code becomes I/O dependent.
- Report errors by throwing an exception. Violates principal of *reserving exceptions for exceptional situations*. Makes an otherwise pure function impure.
- Report errors with some kind of special error return value.
- Combine error and success return values and provide a way of easily continuing the happy path. Again, languages like Haskell provide this.

MongoDb

- One of many nosql databases. No rigid relations need to be predefined.
- Allows storing and querying json documents.
- Provides basic **Create**-**Read**-**Update**-**Delete** (CRUD) repertoire.

Mongo CRUD

All operations asynchronous. Set up to return a `Promise` when called without a handler.

Create	`insertOne()` and `insertMany()`.
Read	`find()` returns a `Cursor`. Can grab all using `toArray()`.
Update	`updateOne()` and `updateMany()`. Also, `findOneAndUpdate()` and `findOneAndReplace()`. Can combine insert and update functionality using `upsert` option.
Delete	`deleteOne()` and `deleteMany()`.

User Store Features

- Store user-info objects.
- No schema for user-info objects, except that each object **must** have an `id` property.
- Have `id` property default to email set in global git configuration for current user.
- Basic CRUD functionality.

Log

```
$ ./index.mjs
usage: index.mjs DB_URL CMD [ARGS]
where CMD [ARGS] is
    load JSON_FILE
    create|read|update|delete [NAME=VALUE...]
$ ./index.mjs mongodb://localhost:27017/users load \
    simpsons.json
$ ./index.mjs mongodb://.../users load simpsons.json
EXISTS: user bart already exists
$ ./index.mjs mongodb://localhost:27017/users read
NO_ID: missing id
$ ./index.mjs mongodb://localhost:27017/users read id=bart
{
  "id": "bart",
  "firstName": "Bart",
  "lastName": "Simpson"
}
```

Log Continued

```
$ ./index.mjs mongodb://localhost:27017/users delete id=bart
$ ./index.mjs mongodb://localhost:27017/users read id=bart
NOT_FOUND: user bart not found
$ ./index.mjs mongodb://... read lastName=Simpson
NO_ID: missing id
$ ./index.mjs mongodb://... read id=homer lastName=Simpson
{
  "id": "homer",
  "firstName": "Homer",
  "lastName": "Simpson",
  "email": "chunkylover53@aol.com"
}
$ ./index.mjs mongodb://... read id=homer lastName=simpson
NOT_FOUND: user homer not found
```

Log Continued

```
$ ./index.mjs mongodb://localhost:27017/users create
{
  "id": "umrigar@binghamton.edu"
}
$ ./index.mjs mongodb://localhost:27017/users update \
    id=umrigar@binghamton.edu firstName=Zerksis
{
  "id": "umrigar@binghamton.edu",
  "firstName": "Zerksis"
}
$ ./index.mjs mongodb://localhost:27017/users read \
   id=umrigar@binghamton.edu
{
  "id": "umrigar@binghamton.edu",
  "firstName": "Zerksis"
}
$
```

Initializing Project

```
$ npm init -y #creates package.json
...
$ npm install mongodb      #old versions of npm required --save
npm notice created a lockfile as package-lock.json...
...
$ ls -a
.gitignore  node_modules  package.json
package-lock.json ...
$
```

Mongo Shell Log

Allows interacting with mongo db. Following log assumes that collection `userInfos` in db `users` is loaded with simpsons data.

```
$ mongo
MongoDB shell version v3.6.3
...
> use users
switched to db users
> db.userInfos.find({})
{ "_id" : "bart", "id" : "bart", ... }
{ "_id" : "marge", "id" : "marge", ... }
{ "_id" : "lisa", "id" : "lisa", ... }
{ "_id" : "homer", "id" : "homer", ... }
> db.userInfos.find({"firstName": "Bart"})
{ "_id" : "bart", "id" : "bart", ... }
> db.userInfos.find({}).length()
4
```

Mongo Shell Log Continued

```
> db.userInfos.deleteOne({"firstName": "Bart"})
{ "acknowledged" : true, "deletedCount" : 1 }
> db.userInfos.find({}).length()
3
> db.userInfos.deleteMany({})
{ "acknowledged" : true, "deletedCount" : 3 }
> db.userInfos.find({}).length()
0f
```

Implementation

index.mjs	Wrapper which dispatches to command-line handling.
user.mjs	Trivial pure user model with artificial validation.
user-services.mjs	User services: use pure model with impure db and external programs (using `git` for email id).
user-store.mjs	Implementation of db operations.
util.mjs	Application errors.
cli.mjs	Command-line handling.