# Table of Contents

# The Promise Handbook

This book serves to teach about promises, justify promises, and show usage patterns.

# The Promise of Promises

## 🚧 Justification

## 🚧 Flow Diagram

# Promise API

## `new Promise()`

- `new Promise((resolve, reject) => {})`

```
const getDinosaur = (name) => {
  return new Promise((resolve, reject) => {
    resolve({name})
  })
}

//Shorthand for synchronous values as a Promise
const getDinosaur = (name) => Promise.resolve({name})
const getDinosaur = (name) => Promise.reject(new Error("Dinosaur
s went extinct!")
```

## `Promise.then`

- Get the asynchronous value

```
getDinosaur("Stegosaurus")
  .then((dinosaur) => console.log(dinosaur))
```

- `.then()` coerces all returned values to a `Promise` (chaining)

```
getDinosaur("Brontosaurus")
  .then((dinosaur) => dinosaur.name)
  .then((name) => console.log(name))
```

## `Promise.catch`

- Handle errors with a `Promise`

```
const getDinosaurs = () => Promise.reject("Connection Timed Out!"
)
getDinosaurs()
  .catch((reason) => console.error(reason))
```

## `Promise.all`

- Asynchronous iteration

```
const getDinosaur = (name) =>
  new Promise((resolve) =>
    setTimeout(() =>
      resolve({name, age: jurrasicPeriod()}),
      Math.random() * 3000)
  )
const jurrasicPeriod = () => Math.floor(Math.random() * 54e6) +
145e6
const log = (value) => console.log(value)
const dinosaurs = ['Brontosaurus', 'Tyrannosaurus', 'Stegosaurus'
]

Promise.all(dinosaurs.map(getDinosaur))
  .then(log) //-> [{name: "Brontosaurus", age: ...}, {name: "Tyr
annosaurus", age: ...}, {name: "Stegosaurus", age: ...}]
```

# Debugging a Promise

## 🚧 Add a `.catch()`

## Chrome `async` toggle

- Get the full stack-trace you expect from a `Promise` chain link

# Promise Conventions for Happier Developers

## 🚧 Name Promise Functions with `getX()`

## Name Inline Functions with verbs

- Improves readability and debugging

```
const getName = (dinosaur) => dinosaur.name
const log = (value) => console.log(value)
getDinosaur()
  .then(getName)
  .then(log)
```

## 🚧 Indent chained methods `.then()` `.catch()`

**SPLIT INTO MULTIPLE PAGES**

## Promise Extensions

### `promise.tap(fulfilled?: Function , rejected?: Function) => Promise`

- A pass-through that gives access to the current value

```
//Warning: Modifies the Promise prototype
Promise.prototype.tap = function (onFulfilled, onRejected) {
  return this.then(
    (result) =>
      onFulfilled
        ? Promise.resolve(onFulfilled(result))
            .then(() => result)
        : Promise.resolve(result)
    ,
    (reason) => {
      onRejected
        ? Promise.resolve(onRejected(reason))
            .then(() => Promise.reject(reason))
        : Promise.reject(reason)
    }
  )
}
const yell = (words) => console.log(`${words.toUpperCase()}!!!`)
const log = (value) => console.log(value)
const logError = (reason) => console.error(reason)

Promise.resolve('Roar!').tap(yell).then(log)
Promise.reject(new Error('Timed Out')).tap(log, logError)
```

### `promise.always(fulfilled?: Function , rejected?: Function) => Promise`

- Do something on settled. (fulfilled or rejected)

```
//Warning: Modifies the Promise prototype
Promise.prototype.always = function (onSettled) {
  return this.then(
    (value) => Promise.resolve(onSettled())
      .then(() => value),
    (reason) => onSettled()
      .then(() => Promise.reject(reason)))
}

takeALongTimeToLoad().always(hideLoadingIndicator)
```

## 🚧 `Promise.Map(values: Array, predicate: Function) => Promise`

## 🚧 `Promise.Reduce(values: Array, predicate: Function, accumulator: Any) => Promise`

## 🚧 `Promise.Filter(values: Array, predicate: Function) => Promise`

**SPLIT INTO MULTIPLE PAGES**

# Promise Patterns for Happier Relationships

## WIP Multiple Dependencies

## Fallback Data

- Insert a different value if a `Promise` fails.

```
const getData = () => Promise.reject(new Error('Connection Timed Out'))
const getCachedData = () => Promise.resolve({name: 'Brontosaurus'})
const log = (value) => console.log(value)
getData()
  .catch(getCachedData) //Could get data from localStorage
  .then(log)
```

- Can be used any time an external resource isn't reliable

## Fanning

- Taking a `Promise` and independently calling multiple `.then()` on it

```
const dinosaurs = getDinosaurs() //Returns a promise

dinosaurs
  .then(cacheDinosaurs)

dinosaurs
  .then(renderDinosaurList)

dinosaurs
  .then(renderDinosaursCount)
```

## Promisify a Callback

- Taking a callback and turning it into a `Promise`

```
const getDinosaur = (name, callback) => callback(null, {name})
const getDinosaurPromise = (name) => {
  return new Promise((resolve, reject) => {
    getDinosaur(name, (error, data) => {
      if (error) {
        return reject(error)
      }
      resolve(data)
    })
  })
}
getDinosaurPromise("velociraptor")
  .then(log) // -> {name: "velociraptor"}
```

## Gate Keeper

- Wrap DOM Ready event

```
// This is how JQuery's $.ready() works
const domReady = () => {
  const readyState = document.readyState
  return readyState === "interactive" || readyState === "complete"
    ? Promise.resolve()
    : new Promise((resolve) =>
      document.addEventListener("DOMContentLoaded", resolve))
}
const roar = () => console.log("Roar!!!")
const attachRoarHandler = () =>
  document.querySelector("button").addEventListener("click", roar);

domReady() // Promise only resolves when the DOM is loaded
  .then(attachRoarHandler)
```

- Can be coupled with fanning to great effect

## Caching

- Store a `Promise` into a cache instead of the values

```
const cache = {}
const getDinosaur = (name) =>
  cache[name]
    ? cache[name]
    : cache[name] = Promise.resolve({name})
const log = (value) => console.log(value)

getDinosaur('Stegosaurus')
  .then(log)
getDinosaur('Stegosaurus')
  .then(log)

console.log(getDinosaur('Stegosaurus') === getDinosaur('Stegosaurus')) //-> true
```

- This way, a hundred different things can request a remote resource and it will only hit it once
    - It does this by storing an unfulfilled `Promise`
- Synchronous values don't populate the cache until the request finished causing all of the requests that come in before the request resolves will also try and hit the external resource

## Throttling

- Return the same `Promise` when something is asked for until it resolves

```
let throttledPromise
const resetThrottle = () => throttledPromise = undefined
const getDinosaurs = () =>
  throttledPromise
    ? throttledPromise
    : throttledPromise = Promise.resolve({name}).always(resetThr
ottle)

const dinosaurs = getDinosaurs()
console.log(dinosaurs === getDinosaurs()) //-> true
setTimeout(() => {
  //By now, the Promise has settled and is no longer pending
  console.log(dinosaurs === getDinosaurs()) //-> false
})
```

- This keeps data fresh, but prevents parallel requests for the same thing
- All parallel requests get the same `Promise` back until it resolves and a new `Promise` is made

## Fastest Promise

- `Promise.race` offers a method to put a time-limit on how long an asynchronous task can take

```
let cache = '[]'

const retrieveDinosaurs = () => //Don't resolve until 5 seconds have passed
  new Promise((resolve) => setTimeout(() => resolve(["Brontosaurus"]), 2000))
const getCachedDinosaurs = () => //Don't resolve until 2 seconds have passed
  new Promise((resolve) => setTimeout(() => resolve(getDinosaursFromCache()), 1000))

const saveDinosaursToCache = (dinosaurs) => cache = JSON.stringify(dinosaurs)
const getDinosaursFromCache = () => JSON.parse(cache)

const getDinosaurs = () => retrieveDinosaurs().tap(saveDinosaursToCache)
const log = (value) => console.log(value)

Promise.race([getDinosaurs(), getCachedDinosaurs()])
  .tap(log)

setTimeout(
  () => Promise.race([getDinosaurs(), getCachedDinosaurs()])
    .tap(log),
  3000)
```

- If a connection is unreliable, take data from the cache and update the cache later

## `async` / `await`

- The future syntax of `Promise`

```
async function getDinosaurs () {
  const dinosaurs = await api.dinosaurs.get() //returns a Promise

  const names = dinosaurs.map((dinosaur) => dinosaur.name)
  return names
}
```

# Glossary

## Promise

An eventual value. An operation that hasn't completed yet, but will in the future.

## Asynchronous

Order of execution isn't strictly in order.

# End

You now have a new tool in your toolbox when coding in JavaScript. Asynchronicity is what makes JavaScript a magical language. Using a `Promise` coupled with the right pattern will increase readability and make tomorrow's applications easier to write.

```
getBook("Promise Handbook")
  .then(readBook)
  .then(digestConcepts)
  .then(practiceConcepts)
  .then(masterPromises)
  .catch(napAndTryAgain)
```

## References

- Kornel Lesiński: And .then() what? Promise programming patterns – Falsy Values 2015
- test double: Common Patterns Using Promises
- ponyfoo: Understanding JavaScript’s async await

## Note From Author

I learned so much in the making of this book about promises. I went from only really understanding the basic concepts of `promise.then` to constructing useful patterns and extensions. I cannot recommend this path enough for anyone wanting to learn about something. Get yourself to write a book about a topic you want to learn more about. My students, colleagues, and partner were all incredibly supportive in going over my edits and providing suggestions.