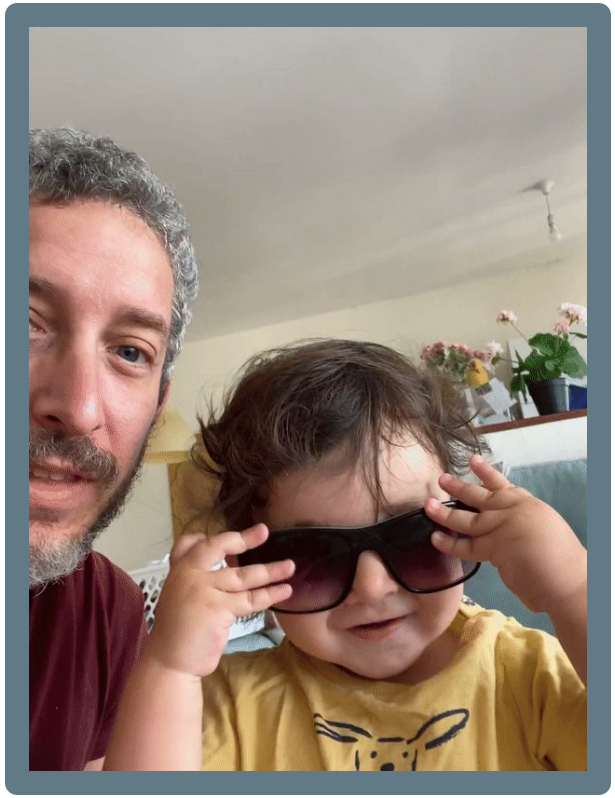
The 5 developers' principles that helped me as a new father

You won't believe #7!



Monolith Design

Spoiler Alert 

Spoiler Alert 

Key takeaways from this talk

1

It's all about decoupling

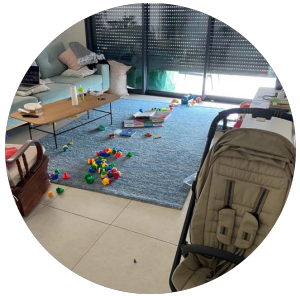
Spoiler Alert 

2

Keep it simple and be kind

Spoiler Alert 

The Principles



Single Responsibility Principle

The Boy Scout Rule

Keep It Simple, Stupid

You Ain't Gonna Need It

Tell, Don't Ask

Single Responsibility Principle

Every class, module, or function should have one responsibility in a program

Single Responsibility Principle

```
async function getUserAndLocation(userId) {
  const user = await getUser(userId);
  // validation and normalization
  const location = await getLocationFromIp(user.ip);
  // validation and normalization
  return {normalizedUser, normalizedLocation}
}
```

Single Responsibility Principle

```
async function getUserAndLocation(userId) {
  const user = await getUser(userId);
  // validation and normalization
  const location = await getLocationFromIp(user.ip);
  // validation and normalization
  return {normalizedUser, normalizedLocation}
}
```

Single Responsibility Principle

```
async function getUserAndLocation(userId) {
  const user = await getUser(userId);
  // validation and normalization
  const location = await getLocationFromIp(user.ip);
  // validation and normalization
  return {normalizedUser, normalizedLocation}
}
```

Single Responsibility Principle

```
async function getUserAndLocation(userId) {
  const user = await getUser(userId);
  // validation and normalization
  const location = await getLocationFromIp(user.ip);
  // validation and normalization
  return {normalizedUser, normalizedLocation}
}
```

Single Responsibility Principle

```
async function getUserAndLocation(userId) {
  const user = await getUser(userId);
  // validation and normalization
  const location = await getLocationFromIp(user.ip);
  // validation and normalization
  return {normalizedUser, normalizedLocation}
}

async function getUser(userId) {
  const user = await getUserFromDb(userId)
  // validation and normalization
  return normalizedUser
}
```

Single Responsibility Principle

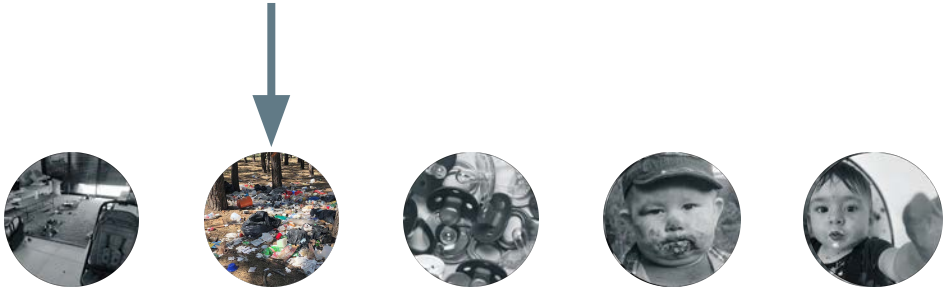
```
async function getUserAndLocation(userId) {
  const user = await getUser(userId);
  // validation and normalization
  const location = await getLocationFromIp(user.ip);
  // validation and normalization
  return {normalizedUser, normalizedLocation}
}

async function getLocation(ip) {
  const user = await getLocationFromIp(ip)
  // validation and normalization
  return normalizedLocation
}
```

Single Responsibility Principle

```
async function getUser(userId) {
  const user = await getUserFromDb(userId)
  // validation and normalization
  return normalizedUser
}

async function getLocation(ip) {
  const user = await getLocationFromIp(ip)
  // validation and normalization
  return normalizedLocation
}

const user = await getUser(userId)
const location = await getLocation(user.ip)
```

The Boy Scout Rule

Leave your code better than you found it



The Boy Scout Rule

```
const DEFALT_DATE = '2022-01-01';
function calculateIncome({row, date = DEFALT_DATE}) {
  return row
            .filter(r => r.date.getDate() >= date)
            .reduce((acc, r) => acc + r.income, 0)
}

// TODO: Add calculateProfit here
```

The Boy Scout Rule

```
const DEFALT_DATE = '2022-01-01';
function calculateIncome({row, date = DEFALT_DATE}) {
  return row
            .filter(r => r.date.getDate() >= date)
            .reduce((acc, r) => acc + r.income, 0)
}

// TODO: Add calculateProfit here
```

The Boy Scout Rule

```
const DEFUALT_DATE = '2022-01-01';
function calculateIncome({row, date = DEFALT_DATE}) {
  return row
            .filter(r => r.date.getDate() >= date)
            .reduce((acc, r) => acc + r.income, 0)
}

// TODO: Add calculateProfit here
```

The Boy Scout Rule

```
const DEFUALT_DATE = '2022-01-01';
function calculateIncome({row, date = DEFALT_DATE}) {
  return row
            .filter(r => r.date.getDate() >= date)
            .reduce((acc, r) => acc + r.income, 0)
}

// TODO: Add calculateProfit here
```

The Boy Scout Rule

```
const DEFUALT_DATE = '2022-01-01';
function calculateIncome({expenses, date = DEFALT_DATE}) {
  return expenses
    .filter(expense => expense.date.getDate() >= date)
    .reduce((totalIncome, expense) => totalIncome + expense.income, 0)
}

// TODO: Add calculateProfit here
```

Keep It Simple, Stupid

A simple solution is better than a complex one

Keep It Simple, Stupid

```
function calculateBySection(section) {
  if (section.type === SECTION_TYPE.REVENUE) {
    // Revenue logic
    return result
  }
  if (section.type === SECTION_TYPE.COGS) {
    // Cost Of Good Sold logic
    return result
  }
  if (section.type === SECTION_TYPE.OPERATIONAL_EXPENSES) {
    // Cost Of Operational Expenses logic
    return result
  }
  if (section.type === SECTION_TYPE.PROFIT) {
    // Cost Of Profit logic
    return result
  }
}

const result = Object.keys(SECTION_TYPE)
    .map(section => ({[section]: calculateBySection(section)}))
```

Keep It Simple, Stupid

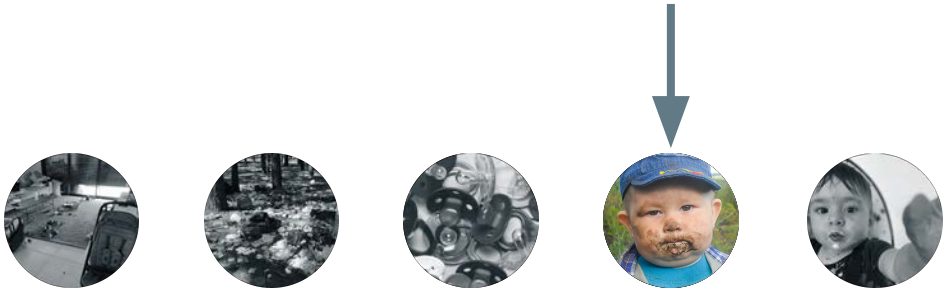
```
function calculateBySection(section) {
  if (section.type === SECTION_TYPE.REVENUE) {
    // Revenue logic
    return result
  }
  if (section.type === SECTION_TYPE.COGS) {
    // Cost Of Good Sold logic
    return result
  }
  if (section.type === SECTION_TYPE.OPERATIONAL_EXPENSES) {
    // Cost Of Operational Expenses logic
    return result
  }
  if (section.type === SECTION_TYPE.PROFIT) {
    // Cost Of Profit logic
    return result
  }
}

const result = Object.keys(SECTION_TYPE)
    .map(section => ({[section]: calculateBySection(section)}))
```

Keep It Simple, Stupid

```
function calculateRevenue() {
  // Revenue logic
}

function calculateCostOfGoodSold() {
  // Cost Of Good Sold logic
}

function calculateOperationalExpenses() {
  // Cost Of OperationalExpenses logic
}

function calculateProfit() {
  // Cost Of Profit logic
}

const result = {
  SECTION_TYPE.REVENUE: calculateRevenue(),
  SECTION_TYPE.COGS: calculateCostOfGoodsSold(),
  SECTION_TYPE.OPERATIONAL_EXPENSES: calculateOperationalExpenses(),
  SECTION_TYPE.PROFIT: calculateProfit()
}
```

You Ain't Gonna Need It

Always implement things when you actually need them, never when you just foresee that you may need them.



You Ain't Gonna Need It

```
class BaseRepository {
  get(id) {
    throw new Error('Method not implemented.');
  }
}

class UserRepository extends BaseRepository {
  #client
  constructor() {
    this.#client = new PrismaClient();
  }
  async get(id) {
    return this.#client.user.findUnique({ where: { id } });
  }
}
```

You Ain't Gonna Need It

```
class BaseRepository {
  get(id) {
    throw new Error('Method not implemented.');
  }
}

class UserRepository extends BaseRepository {
  #client
  constructor() {
    this.#client = new PrismaClient();
  }
  async get(id) {
    return this.#client.user.findUnique({ where: { id } });
  }
}
```

You Ain't Gonna Need It

```
class UserRepository {
  #client
  constructor() {
    this.#client = new PrismaClient();
  }
  async get(id) {
    return this.#client.user.findUnique({ where: { id } });
  }
}
```

You Ain't Gonna Need It

```
class UserRepository {
  #client
  constructor() {
    this.#client = new PrismaClient();
  }
  async get(id) {
    return this.#client.user.findUnique({ where: { id } });
  }
}
```

You Ain't Gonna Need It

```
class UserRepository {
  #client
  constructor() {
    this.#client = await db.get()
  }
  async get(id) {
    const query = 'SELECT * FROM users WHERE id=%d'
    const [user] = await this.#client.query(query, id)
    return user
  }
}
```

You Ain't Gonna Need It

```
class UserRepository {
  #client
  constructor() {
    this.#client = await db.get()
  }
  async get(id) {
    const query = 'SELECT * FROM users WHERE id=%d'
    const [user] = await this.#client.query(query, id)
    return user
  }
}
```

You Ain't Gonna Need It

```
async function getUser(id) {
  const client = await db.get()
  const query = 'SELECT * FROM users WHERE id=%d'
  const [user] = await this.#client.query(query, id)
  return user
}
```

You Ain't Gonna Need It

```
class BaseRepository {
  get(id) {
    throw new Error('Method not implemented.');
  }
}

class UserRepository extends BaseRepository {
  #client
  constructor() {
    this.#client = new PrismaClient();
  }
  async get(id) {
    return this.#client.user.findUnique({ where: { id } });
  }
}
```

You Ain't Gonna Need It

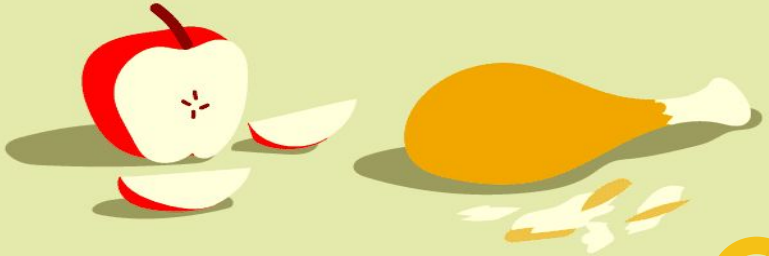
```
async function getUser(id) {
  const client = await db.get()
  const query = 'SELECT * FROM users WHERE id=%d'
  const [user] = await this.#client.query(query, id)
  return user
}
```

The Stages of Baby Food

Stage 1: 4 to 6 Months Old
Single ingredient foods and pureed fruits and veggies

Stage 2: 7 to 8 Months Old
Strained food combinations for new tastes and textures

Stage 3: 9 to 12 Months Old
Foods in small chunks that must be chewed



Stage 4: Over 12 Months Old
Table food can be introduced

Tell,
Don't Ask

Rather than asking for data and acting on that data, we should instead tell it what to do.

Tell, Don't Ask

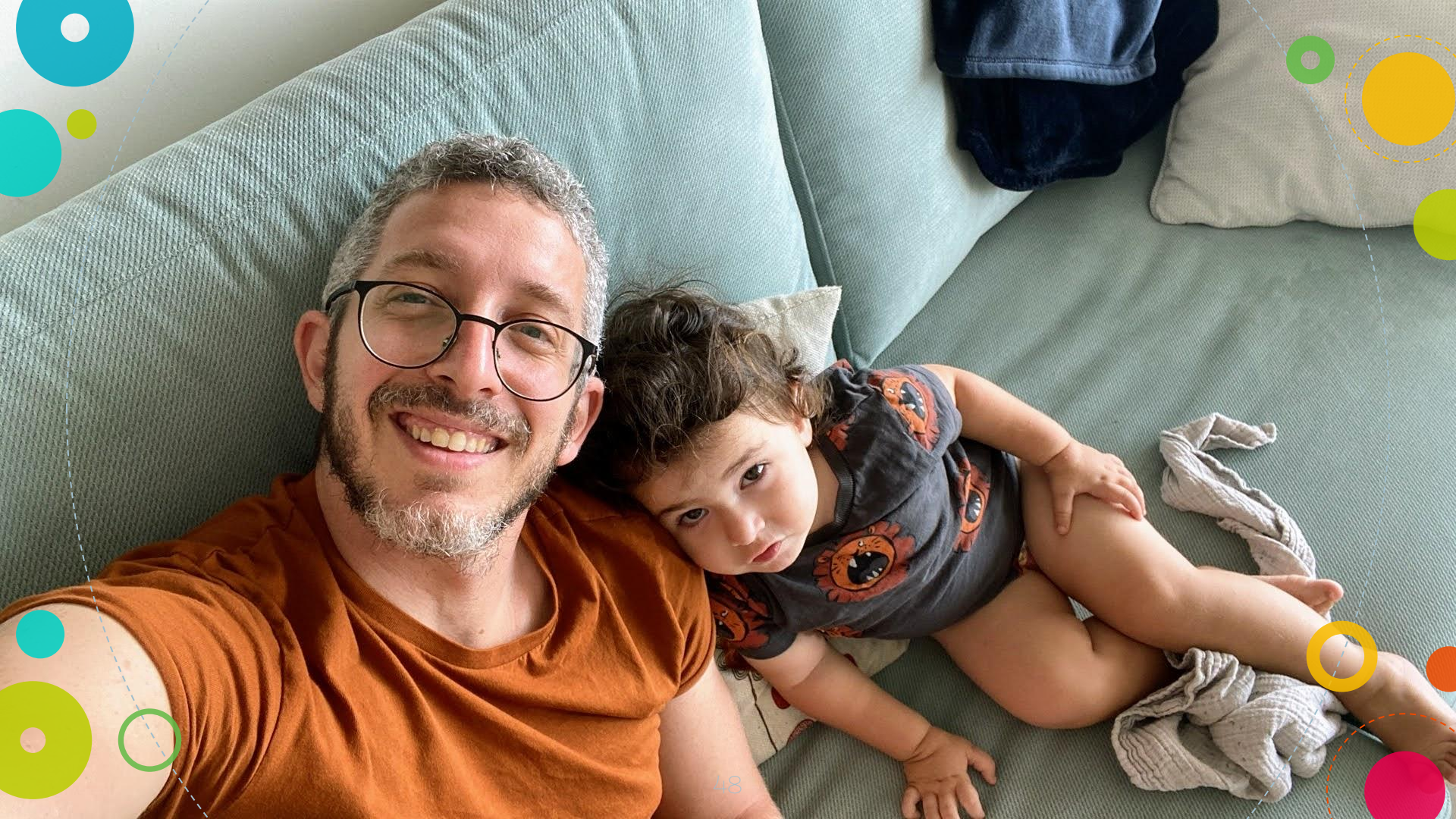
```
const maxDate = new Date(2022,1,1);
if (task.status === Status.NOT_STARTED &&
    task.createdAt < maxDate &&
    task.subscribers.isEmpty()) {
        task.close()
}

class Task {
  // ...
  // Task related code
  //...
  close() {
    this.status = Status.RESOLVED
  }
}
```

Tell, Don't Ask

```
const maxDate = new Date(2022,1,1);
if (task.status === Status.NOT_STARTED &&
    task.createdAt < maxDate &&
    task.subscribers.isEmpty()) {
        task.close()
}

class Task {
  // ...
  // Task related code
  //...
  close() {
    this.status = Status.RESOLVED
  }
}
```

Tell, Don't Ask

```
const maxDate = new Date(2022,1,1);
task.closeIfUnclaimed({createdBefore: maxDate})

class Task {
  // ...
  // Task related code
  //...
  closeIfUnclaimed({ createdBefore }) {
    if (self.status === Status.NOT_STARTED &&
        self.createdAt < createdBefore &&
        self.subscribers.isEmpty()) {
            self.task = Status.RESOLVED
        }
    }
}
```



Thanks!

You can find me at:

@benytol

tb.tlusty@gmail.com