## **1. Overview**

Defensive coding ensures that your application remains reliable and secure even under adverse conditions, such as erroneous inputs or malicious attacks. Two complementary strategies apply:

- Securing Programming: Prevent issues by validating input, sanitizing data, and designing with security in mind.

- Offensive Programming: Strengthen code with runtime checks, guard clauses, and clear failure modes.

Throughout this document, we will explore various techniques to bolster your applications, including:

1. Input/Output Validation & Sanitization

2. Assertions & Guard Clauses

3. Handling Null/Undefined

4. Protecting Properties & State

5. Thorough Exception Handling

6. Comprehension & Simplicity, Observability

7. Boundary Checks

8. Testing

9. Secure Programming

10. Designing Alternative Workflows

11. Static Code Analysis

12. Code Review

13. Chaos Engineering

14. Defensiveness Metrics

## **2. Input/Output Validation & Sanitization**

### **2.1 Why It Matters**

Unvalidated inputs are the most common source of bugs and security holes (e.g., XSS, SQL injection). Likewise, failing to sanitize outputs can introduce vulnerabilities when rendering user data.

### **2.2 Validation**

- Fail Fast: Validate inputs as soon as they come in. If invalid, throw or return an error immediately.

- Guard Clauses: Use small condition checks at the start of a function to confirm the data meets expected constraints.

Example with `validator.js`:

```js

import validator from 'validator';

function processEmail(inputEmail) {

const trimmedEmail = validator.trim(inputEmail);

if (!validator.isEmail(trimmedEmail)) {

throw new Error('Invalid email address');

}

return validator.normalizeEmail(trimmedEmail);

}

**Example with express-validator:**

const { body, validationResult } = require('express-validator');

app.post('/register', [

body('email').isEmail(),

body('password').isLength({ min: 8 })

], (req, res) => {

const errors = validationResult(req);

if (!errors.isEmpty()) {

return res.status(400).json({ errors: errors.array() });

}

// Proceed with valid data...

});

### **2.3 Sanitization**

* **DOM Injection**: Use a library like [DOMPurify](https://www.npmjs.com/package/dompurify) or [sanitize-html](https://www.npmjs.com/package/sanitize-html) to safely render user-provided HTML (e.g., in React’s dangerouslySetInnerHTML or Angular’s [innerHTML]).
* **SQL Injection**: In Node.js, use parameterized queries or safe ORM solutions (e.g., Sequelize, TypeORM, Prisma).

### **2.4 Recommended Libraries**

* [**validator.js**](https://www.npmjs.com/package/validator) for string validation & sanitization.
* [**Joi (now 'hapi/joi')**](https://www.npmjs.com/package/joi) for schema-based validation.
* [**express-validator**](https://www.npmjs.com/package/express-validator) for Express route-level validation.
* [**AJV**](https://www.npmjs.com/package/ajv) for JSON schema validation.
* [**DOMPurify**](https://www.npmjs.com/package/dompurify) or [**sanitize-html**](https://www.npmjs.com/package/sanitize-html) for HTML sanitization.
* [**Zod**](https://www.npmjs.com/package/zod) for runtime type checking and schema validation in TypeScript.

## **3. Assertions & Guard Clauses**

### **3.1 Why They Matter**

Assertions and guard clauses ensure that **key assumptions** about your program (like input validity or object shape) are enforced at runtime. Failing fast when assumptions break makes bugs more visible and prevents deeper corruption.

### **3.2 Guard Clauses**

Use **guard clauses** to quickly exit a function if a condition is not met:

function getUser(id: string) {

if (!id) {

throw new Error('User ID must not be empty');

}

// proceed...

}

### **3.3 Assertions**

**Assertions** can be implemented in a few ways:

**TypeScript Assertion Functions**:  
  
 function assertString(value: unknown): asserts value is string {

if (typeof value !== 'string') {

throw new Error('Expected a string');

}

}

function processInput(value: unknown) {

assertString(value);

// Here, value is known to be a string...

}

**Node.js assert module**:  
  
 const assert = require('assert');

function calculateDiscount(price) {

assert(price >= 0, 'Price must be non-negative');

// ...

}

**Runtime Validation Libraries**:  
 Use libraries like [**zod**](https://www.npmjs.com/package/zod), [**runtypes**](https://www.npmjs.com/package/runtypes), [**io-ts**](https://www.npmjs.com/package/io-ts) to enforce runtime types or constraints:  
  
 import \* as z from 'zod';

const UserSchema = z.object({

name: z.string(),

age: z.number().min(0).max(120),

});

function createUser(data: unknown) {

const parsed = UserSchema.parse(data); // throws if invalid

// ...

}

Using **assertions** clarifies your expectations and ensures that misuse of your code triggers a clear error rather than a silent failure.

## **4. Handling Null/Undefined**

### **4.1 Why It Matters**

null and undefined are common sources of runtime errors in JavaScript. Defensive strategies eliminate or safely handle them.

### **4.2 Defensive Patterns**

**Check for Nullish**: Guard against null or undefined before accessing properties.  
  
 function getUserName(user?: { name?: string }) {

if (!user || !user.name) {

throw new Error('User or user.name is missing');

}

return user.name;

}

**Optional Chaining**: Use obj?.prop?.subProp to avoid runtime errors on deeply nested objects.  
  
 const username = user?.profile?.name ?? 'Unknown';

**Non-Nullable Types**: Enable strictNullChecks in your tsconfig.json. Declare fields as non-nullable if you are certain they must exist.  
  
 // Example: With strictNullChecks, this line raises an error

// if 'user' could be null or undefined.

function printUser(user: { name: string }) {

console.log(user.name);

}

1. **Runtime Null Checking Libraries**:  
   * **Joi** or **Zod** can also be used to ensure a value is not null or undefined before proceeding.

### **4.3 Example**

function processAge(age?: number) {

if (age == null) {

throw new Error("age can't be null or undefined");

}

// Now 'age' is guaranteed to be a valid number

if (age < 0 || age > 120) {

throw new RangeError('Age out of range');

}

// ...

}

By proactively checking and clarifying **null**/**undefined** usage, you greatly reduce unexpected TypeError: Cannot read property ... crashes.

## **5. Protecting Properties & State**

### **5.1 Immutability and Controlled Access**

**Why**: Minimizing unintended mutations helps prevent bugs and security holes.

1. **Immutable Data**
   * In **React**, treat props and state as immutable; never mutate them directly.
   * Consider libraries like [Immer](https://immerjs.github.io/immer/) or [immutable.js](https://immutable-js.com/).
2. **Encapsulation**
   * Hide direct property access behind class methods or closures.
   * Use TypeScript’s private or readonly for compile-time checks.

**Example**:

class UserProfile {

private \_email: string;

constructor(email: string) {

this.\_email = email;

}

get email(): string {

return this.\_email;

}

set email(newEmail: string) {

// Insert validation/sanitization here

this.\_email = newEmail;

}

}

### **5.2 Angular Services & RxJS**

* **Share state** via BehaviorSubject but expose only read-only streams to components.
* **Avoid Global Services** that hold too much shared state—break services down by domain.

### **5.3 React State Management**

* **Redux**: Keep reducers pure, sanitize data before storing.
* **Context**: Split contexts by domain to maintain simplicity.
* **Avoid Over-Mutating**: Overly complex state leads to confusion and security holes.

### **5.4 Node.js Application State**

* **Statelessness**: Typically scale Node.js horizontally; keep minimal in-memory data.
* **Configuration**: Validate environment variables (process.env) at startup.
* **In-Memory Caches**: Use TTL and size limits. Consider external solutions (Redis, Memcached) with proper authentication.

## **6. Thorough Exception Handling**

### **6.1 Try/Catch**

Use try/catch for **risky operations** such as file I/O, network calls, or JSON parsing.

try {

const data = JSON.parse(userInput);

// ...

} catch (error) {

console.error('Failed to parse JSON:', error);

throw new Error('Invalid input format');

}

### **6.2 Custom Error Classes**

Create specialized error classes for clarity:

class ValidationError extends Error {

constructor(message: string) {

super(message);

this.name = 'ValidationError';

}

}

### **6.3 Logging & Error Propagation**

* Use structured logging (e.g., **winston**, **pino**, **bunyan**).
* Decide carefully when to handle errors locally vs. rethrow them up the stack.

## **7. Comprehension & Simplicity, Observability**

### **7.1 Comprehension & Simplicity**

1. **Small, Focused Functions**: Keep them short, do one thing.
2. **Consistent Coding Standards**: Use **ESLint** + **Prettier**.
3. **Avoid Clever Code**: Aim for clarity, not “tricks.”
4. **Principle of Least Surprise**: Follow common patterns.

### **7.2 Observability**

1. **Structured Logging**: Log in JSON with contextual metadata (e.g., request IDs).
2. **Metrics**: Expose performance counters or custom metrics (Prometheus, prom-client).
3. **Distributed Tracing**: Implement solutions like OpenTelemetry, Jaeger, or Zipkin.
4. **Health Checks & Alerts**: Provide /health endpoints, set up alerting for failures.

## **8. Boundary Checks**

Keep data within expected ranges:

* **Length Checks**: if (username.length > 50) throw ...
* **Numeric Range Checks**: if (price < 0 || price > 10000) throw ...
* **Enum Validations**: If the parameter must be from a known set, explicitly check it.

**Example**:

function processAge(age: number) {

if (age < 0 || age > 120) {

throw new RangeError('Age must be between 0 and 120');

}

// ...

}

## **9. Testing**

### **9.1 Unit Testing**

* Use frameworks like [**Jest**](https://jestjs.io/), [**Mocha**](https://mochajs.org/), or [**Vitest**](https://vitest.dev/).
* Test edge cases and error scenarios as well as the happy path.

test('throws if invalid email', () => {

expect(() => processEmail('not-an-email')).toThrow();

});

### **9.2 Integration Testing**

* Validate module interactions, e.g., full route tests in Express or e2e tests in Angular/React.
* Test with real or mock databases and network calls.

### **9.3 Security Testing**

* **ZAP** (Zed Attack Proxy) for dynamic testing.
* **npm audit** or **yarn audit** for vulnerability scans.

## **10. Secure Programming**

### **10.1 HTTP Headers**

* Use [**Helmet**](https://www.npmjs.com/package/helmet) in Express to set secure HTTP headers.

### **10.2 CORS**

* Be strict with cross-origin resource sharing. Use the [**cors**](https://www.npmjs.com/package/cors) package to whitelist domains.

### **10.3 Secrets Management**

* Never commit secrets to source control.
* Use environment variables or a vault solution.
* Validate the presence and format of secrets at startup.

### **10.4 Dependency Management**

* **npm audit** or **yarn audit** to detect known vulnerabilities.
* Keep dependencies updated, remove unused ones.

## **11. Designing Alternative Workflows**

**Designing alternative workflows** involves **planning and implementing backup paths, fallbacks, and recovery strategies**—critical for resiliency and graceful degradation.

### **11.1 Why Alternative Workflows Matter**

Systems fail. A well-thought-out fallback or backup path:

* Improves **user experience** by avoiding full outages.
* Ensures partial functionality can continue.
* Allows time for automated or manual recovery.

### **11.2 Backup Paths**

A **backup path** is a secondary route if the primary fails.

* **Front-End Example**: If fetching personalized data fails, show default/popular items.
* **Node.js Example**: If logging to a central server fails, write logs locally or to a backup endpoint.

async function fetchRecommendations(userId: string): Promise<Item[]> {

try {

// primary

return await fetchFromService(`/api/recommendations/${userId}`);

} catch (error) {

// backup path

console.warn(`Falling back due to error: ${error}`);

return fetchFromService('/api/popular-items');

}

}

### **11.3 Fallbacks**

**Fallbacks** return default or cached data when a service is unavailable:

1. **Cached Data Fallback**: Return the last-known valid response from Redis or in-memory cache.
2. **Stubbed Response**: Provide minimal placeholders.
3. **Feature Toggle**: Temporarily disable a non-critical feature.

async function getUserProfile(userId: string) {

try {

return await httpClient.get(`/user-profile/${userId}`);

} catch {

console.warn('Primary service failed, fallback to cache');

const cachedProfile = await redisClient.get(`profile:${userId}`);

if (cachedProfile) return JSON.parse(cachedProfile);

throw new Error('No profile available, fallback also failed');

}

}

### **11.4 Recovery Strategies**

* **Retries with Exponential Backoff**: Re-attempt failed requests after progressive delays ([**p-retry**](https://www.npmjs.com/package/p-retry)).
* **Circuit Breakers**: Detect repeated failures, “open” the circuit to prevent further calls ([**opossum**](https://www.npmjs.com/package/opossum)).
* **Graceful Shutdown & Auto-Restart**: Use [**PM2**](https://pm2.keymetrics.io/) or Docker/Kubernetes to automatically restart crashed processes.

**Circuit Breaker Example**:

import CircuitBreaker from 'opossum';

const options = { timeout: 3000, errorThresholdPercentage: 50, resetTimeout: 10000 };

const breaker = new CircuitBreaker(async () => {

return await httpClient.get('/some-service');

}, options);

breaker.fallback(() => ({ data: "Fallback data" }));

### **11.5 Final Example**

import express from 'express';

import CircuitBreaker from 'opossum';

const app = express();

const breakerOptions = { timeout: 2000, errorThresholdPercentage: 50, resetTimeout: 5000 };

const analyticsBreaker = new CircuitBreaker(async (userId: string) => {

return await fetch(`https://analytics.example.com/user/${userId}`).then(res => res.json());

}, breakerOptions);

analyticsBreaker.fallback(async (userId: string) => {

const cached = getFromCache(userId);

if (cached) return cached;

return { visits: 0, lastLogin: null };

});

app.get('/user-analytics/:userId', async (req, res) => {

try {

const data = await analyticsBreaker.fire(req.params.userId);

res.json({ success: true, data });

} catch (error) {

console.error('Analytics retrieval failed:', error);

res.status(500).json({ success: false, message: 'Cannot retrieve analytics at this time.' });

}

});

app.listen(3000, () => console.log('Server on port 3000'));

## **Putting It All Together: Sample Express Snippet**

import express from 'express';

import helmet from 'helmet';

import validator from 'validator';

import { body, validationResult } from 'express-validator';

const app = express();

app.use(express.json());

app.use(helmet()); // Basic security headers

app.post('/create-user', [

body('email').isEmail(),

body('age').isInt({ min: 0, max: 120 })

], (req, res) => {

const errors = validationResult(req);

if (!errors.isEmpty()) {

return res.status(400).json({ errors: errors.array() });

}

const { email, age } = req.body;

// Additional sanitization

const sanitizedEmail = validator.normalizeEmail(email);

// Boundary check

if (Number(age) < 18) {

return res.status(403).json({ error: 'Minimum age is 18' });

}

try {

// ... create user in DB, etc.

return res.status(201).json({ message: 'User created successfully.' });

} catch (err) {

// Thorough exception handling

console.error('Error creating user:', err);

return res.status(500).json({ error: 'Internal server error' });

}

});

app.listen(3000, () => {

console.log('Server running on port 3000');

});

In this snippet:

* **Helmet** secures HTTP headers.
* **Validation & Sanitization** on email and age.
* **Fail Fast** approach returns 400 for bad data.
* **Boundary Check** ensures valid age range.
* **Exception Handling** logs and returns an error response.