Type-Driven Domain Modeling

Use the types, Luke!

Security Meetup – 2022-01-26
Introduction

Wisdoms from the Blue Book
Domain-Driven Design (DDD)

- Code and structure should match the business domain
- Primary focus on core domain and business logic
- Iteratively refine concepts by consulting domain experts
- Uses ubiquitous language that everyone in the domain understands
Popular Concepts

- Entity
- Value object
- Aggregate
- Bounded Contexts
- Repository
- Command Query Responsibility Segregation

Source: Domain-Driven Design – Eric Evans
Type-Driven Domain Modeling

- Encode business rules into types
- Make illegal state unrepresentable
- Reason without looking at implementations
- Prevent security vulnerabilities
- Offload work to the compiler
Language Support

- Any language can be used for DDD
- But some provide more powerful type systems
- Nice to have: Sum Types, Pattern Matching

Examples:
- Haskell
- Elm
- OCaml
- F#
- Rust
- Scala
- Reason
- TypeScript
- C# (7)
- Java (17)
Null

“I call it my billion-dollar mistake. It was the invention of the null reference in 1965.”
Tony Hoare
Make the Absence of Values Explicit (Haskell)

1. data Maybe a = Nothing | Just a

2. lookup :: String -> [(String, String)] -> Maybe String

3. lookup key list = case list of

4.     [] -> Nothing

5.     ((k, v):rest) -> if key == k then Just v else lookup key rest

6. phonebook :: [(String, String)]

7. phonebook = [("Alice", "01889 985333"), ("Bob", "01788 665242")]

8. getNumber :: String -> [(String, String)] -> String

9. getNumber name list = case lookup name list of

10.    Nothing -> "Could not find a number for " <> name

11.    Just value -> name <> "," <> value
Modeling a Contact Type
Because not everything is a string
Demo Time

Domain modeling in F#
Typed Security
Preventing vulnerabilities by design
OWASP Top Ten (2021)

- A01:2021-Broken Access Control
- A03:2021-Injection
- A04:2021-Insecure Design

Source: https://owasp.org/Top10/
OWASP API Security Top 10 (2019)

• API1:2019 Broken Object Level Authorization
• API3:2019 Excessive Data Exposure
• API5:2019 Broken Function Level Authorization
• API6:2019 Mass Assignment
• API8:2019 Injection
Demo Time

Fighting broken access control attacks

SBA Research
Fighting Injection Attacks (Haskell)

1. import qualified Database.SQLite.Simple as SQL

2. main = SQL.withConnection "products.db" $ \conn -> do
3.    putStrLn "Search by product name:"
4.    pname <- getLine
5.    products <- getProductsByName conn pname
6.    putStrLn ("Here is the data: " ++ show products)

7. -- SQL.query :: SQL.Connection -> SQL.Query -> args -> IO [result]

8. getProductsByName :: SQL.Connection -> String -> IO [Product]
9. getProductsByName conn pname =
10.  SQL.query conn (SQL.Query "SELECT * FROM products WHERE product_name=?") (pname)
1. `public class User {`
2. `private final Id<User> id;`
3. `private Name50 firstName, lastName;`
4. `private Birthdate dateOfBirth;`
5. `private PasswordHash passwordHash;`
6. `// Constructor, getters, setters, domain logic, etc.`
7. `}`
8. `public class UserViewModel {`
9. `private final String fullName;`
10. `private final DateTime dateOfBirth;`
11. `// Constructor, getters, mapping from entity to dto and vice versa`
12. `}`
13. `public List<UserViewModel> getUsers(...) {`
14. `List<User> users = userService.getUsers(...);
15. `return users.stream().map(UserViewModel::entityToViewModel).collect(Collectors.toList());`
16. `}`
Fighting XSS Attacks (Elm)

1. -- div : List (Attribute msg) -> List (Html msg) -> Html msg
2. -- text : String -> Html msg

3. userComponent : String -> Int -> Html msg
4. userComponent userName userAge =
5.   div [ class "user-component" ]
6.     [ text userName
7.     , text (String.fromInt userAge)
8.     ]

9. -- <div class="user-component">Foo42</div>
Choosing the Right™ Architecture

Building on SOLID principles
Hexagonal / Onion / Clean Architecture

Source: https://www.4soft.de/blog/2019/clean-architecture/
Example Repository: PackIT (C#)

Source: https://www.youtube.com/watch?v=NzcZcim9tp8
Book Recommendations

Domain Modeling Made Functional
Tackle Software Complexity with Domain-Driven Design and F#

Implementing Domain-Driven Design

Scott Wlaschin

Vaughn Vernon
Foreword by Eric Evans
Key Takeaways

• Make illegal state unrepresentable
• Encode business rules in your types
• Parse, don’t validate
• Use the compiler to your advantage
• Write readable code for domain experts
• Eliminate security vulnerabilities by design