

Domain Objects

CS 346 Application Development

<https://student.cs.uwaterloo.ca/~cs346>

Contents

Introduction	2
Motivation	3
Requirements	5
Data Classes	6
Data Formats	9
Choosing a Format	10
Comma-Separated Values (CSV)	11
Extensible Markup Language (XML)	14
Yet-Another-Markup-Language (YAML)	18
JavaScript Object Notation (JSON)	19
Serialization	21
What is it?	22
Bibliography	28

Introduction

Motivation

Recall our architectural layers. We know how to build a user interface, so we need to shift to understanding how to move data between the UI and the data layer (where data originates). [1]

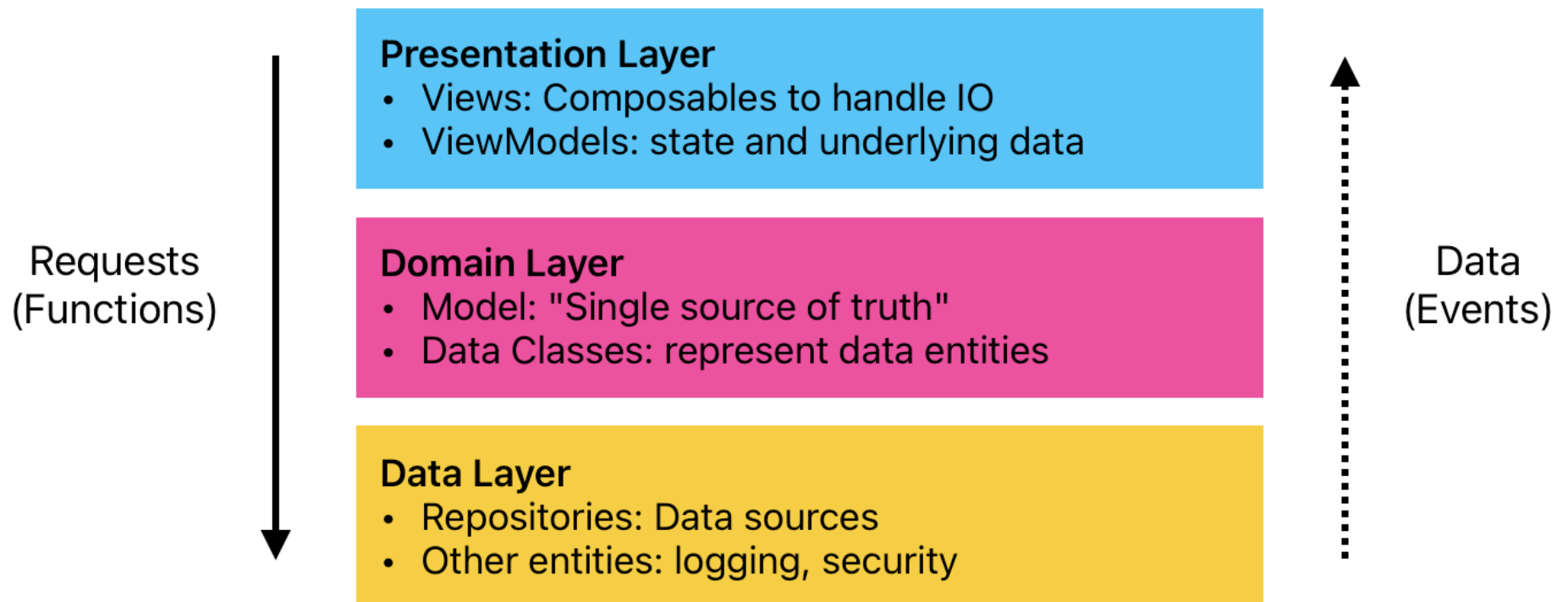


Figure 1: The domain layer mediates between the presentation and data layers.

Motivation

Model

- The single class (`Model.kt`) that stores your application state. The model is the “source-of-truth” for your application.
- Subject in the Observer pattern, it notifies `ViewModels` when state changes.
- Addresses top-level functions for your use cases. Collects data from multiple sources. e.g.,
 - if you have a use case to display customer sales data, your `Model` would pull in data from both the `CustomerRepository` and `SalesRepository` (data layer), and return it to the `ViewModels` that wish to use that data.

Data Classes

- Reflect all of your data entities. e.g., `Recipe` and `Customer` classes.
- Other layers use these data classes for all data representation. e.g., the `CustomerRepository` could load data into a `List<Customer>` to return to the `Model`, which in turn might manipulate that list before returning it to the `CustomerViewModel`.

Requirements

Types of data:

- Primary data: data critical to purpose e.g., graphics editors & images.
- Secondary: user preferences, credentials, API tokens for logging in.

Questions to address:

- What repository manages it? Do I need to consolidate multiple sources?
- Does it have a format that I need to be able to manage? e.g., JPEG images.
- Do I need to cache data locally, or do I reload it as-needed?
- Is this user data (e.g., password), or application data (e.g., window size).
- What operations can I perform. Can I update the repository or is it read-only?
- What are the privacy and security implications of transmitting or storing this data?

Goals

- Data integrity (always).
- Flexibility to manipulate data for different purposes e.g., display, printing.

Data Classes

The obvious choice for storing data in memory are Kotlin [data classes](#). We can use a data class instance to represent a logical record, and store sets of records in regular collections.

```
data class Course (  
    val term: Int,  
    val courseID: String,  
    val subjectCode: String,  
    val catalogNumber: String,  
    val title: String = "",  
    val description: String = ""  
)  
  
val courses = List<Course>()  
courses.add(  
    Course(1251, "01687", "CS", "346")  
)
```

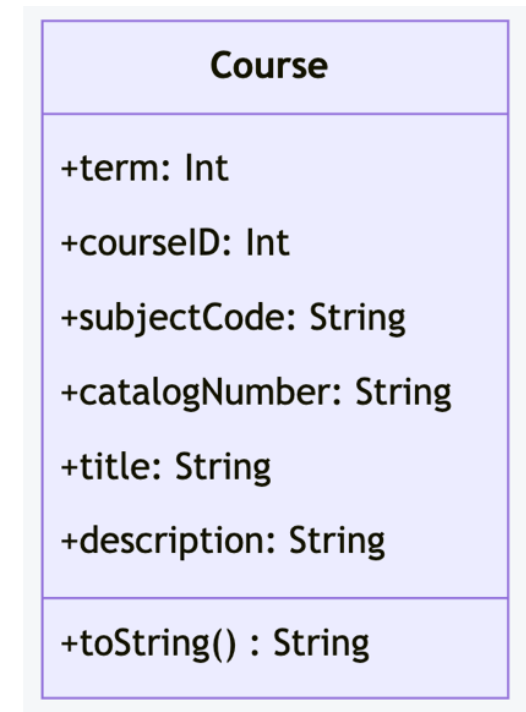


Figure 2: Corresponding class diagram.

Data Classes

Multi-Format Data Classes

You may need customized versions of your data classes that are fit-to-purpose.

```
open class CourseDao(  
    val courseId: String,  
    val courseOfferNumber: Int,  
    val termCode: Int,  
    val termName: String,  
    val subjectCode: String,  
    val catalogNumber: String,  
    val title: String,  
    val description: String,  
    val gradingBasis: String,  
    val courseComponent: String,  
    val enrollConsentCode: String,  
    val enrollConsentDesc: String,  
    val dropConsentCode: String,  
    val dropConsentDesc: String,  
    val requirementsDesc: String?  
)
```

```
open class Course(  
    val term: Int,  
    val courseId: String,  
    val subject: String,  
    val catalogNumber: String,  
    val title: String,  
    val description: String  
)
```

e.g., a web service might return a record with many fields, but you might only need a subset of those fields for your database.

Data Classes

Conversion Between Formats

You will need to convert a record between different representations.

```
open class Course(  
    val term: Int,  
    val courseID: String,  
    val subject: String,  
    val catalogNumber: String,  
    val title: String,  
    val description: String  
) {  
    constructor(course: CourseDao): this(  
        term = course.termCode,  
        courseID = course.courseId,  
        subject = course.subjectCode,  
        catalogNumber = course.catalogNumber,  
        title = course.title,  
        description = course.description,  
    )  
}
```

Data Formats

Choosing a Format

So, internally we store data in data objects. This is great if all you are doing is manipulating Kotlin objects.

However, at some point you need to manipulate data outside of the application.

- How do you get the raw data that you use to create data objects?
- How do you save a data object to a file?
- How do you transmit it over a network?
- How do you save it to a database?

In other words, we need to address the “edges” of our application that involve importing and exporting data to and from data objects. Typically this will mean converting to a different format that the receiver can manage.

We'll discuss a few options, with the goal of choosing a format that is:

- portable across systems,
- easy to debug and human readable.

Comma-Separated Values (CSV)

The simplest way to store records might be to use a [Comma-Separated Values \(CSV\)](#) in plain-text.

We use this structure:

- Each record is stored as UTF8 text with a CRLF terminating the record.
- Values within a record are comma separated.

For example, transaction data stored in a comma-delimited file would look like this:

Customer.csv

```
1001, Jeff Avery, Cambridge
1002, Allison Barnett, Waterloo
1003, John McAfee, Delphi
```

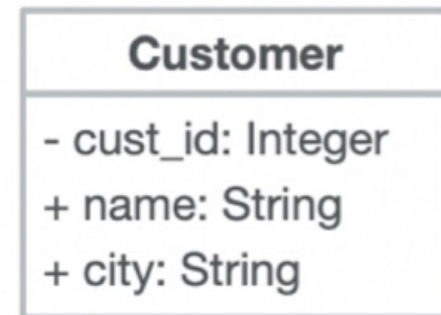


Figure 3: Customer class diagram

Comma-Separated Values (CSV)

Reading/Writing

Reading and writing to a CSV file is trivial, using `stdlib` functions.

```
data class Customer (  
    val cust_id: Int, val name: String, val city: String  
)  
  
val customers = List<Customer>()  
customers.add(Customer(1001, "John Hall", "New York"))  
customers.add(Customer(1002, "Allison Barnett", "Waterloo"))  
customers.add(Customer(1003, "John McAfee", "Delphi"))  
  
File("output.csv").open("w").use {  
    for (customer in customers) {  
        it.write(  
            "${customer.cust_id},${customer.name},${customer.city}\n"  
        )  
    }  
}
```

Comma-Separated Values (CSV)

Why Not CSV?

CSV is literally the simplest possible thing that we can do.

As a file format, it has some advantages:

- Programming languages can easily work with CSV files (they're just text!)
- It's pretty space efficient.
- It's human-readable. Kind-of.

However, CSV comes with some big disadvantages:

- It doesn't work very well if your data contains the delimiter (e.g. a comma in your city field).
- It assumes a fixed structure and doesn't handle variable length records.
- It's hard to read! There is no semantic information to make sense of it. (i.e., there is no simple way to interpret the structure, no schema file format).
- It doesn't work for complex, multi-dimensional data. e.g. Customer transactions.

Extensible Markup Language (XML)

[Extensible Markup Language \(XML\)](#) is a markup language that designed for data storage and transmission.

Defined by the World Wide Web Consortium's XML specification, it was the first major standard for markup languages.

- It's structurally similar to HTML, with a focus on data transmission.
- The structure consists of pairs of tags that enclose data elements. Attributes are optional.

```
<id>2001</id>  
<name>Jeff</name>  
<city>Cambridge</city>  
My portrait.</img>
```

Extensible Markup Language (XML)

Example of a music collection structured in XML.

Taken from the [Chinook DB](#).

```
<catalog>
  <album>
    <title>Empire Burlesque</title>
    <artist>Bob Dylan</artist>
    <company>Columbia</company>
    <price>10.90</price>
    <year>1985</year>
  </album>
  <album>
    <title>Innervisions</title>
    <artist>Stevie Wonder</artist>
    <company>The Record Plant</company>
    <price>9.90</price>
    <year>1973</year>
  </album>
</catalog>
```

Extensible Markup Language (XML)

Reading/Writing

Writing XML is relatively simple. Reading it is not trivial, and requires a [more sophisticated parser](#).

```
data class Customer (  
    val cust_id: Int, val name: String, val city: String  
)  
val customers = List<Customer>() // list of customers
```

```
File("output.xml").open("w").use {  
    it.write("<customers>")  
    for (customer in customers) {  
        it.write("<customer>")  
        it.write("<cust_id>${customer.cust_id}</cust_id>")  
        it.write("<name>${customer.name}</name>")  
        it.write("<city>${customer.city}</city>")  
        it.write("</customer>")  
    }  
    it.write("</customers>")  
}
```

Extensible Markup Language (XML)

Why not XML?

XML provides structure.

- The use of tags, and the optional use of a schema file, means that we can formally define the semantic structure of our data!
- This provides some major advantages compared to CSV.
 - Can rely on structure to infer the meaning of data.
 - You can nest elements e.g., collections of records.

XML is rarely used except in legacy systems. Why?

- Tags “bloat” the data, which results in excessive space requirements.
- Practically impossible to parse without a complex library.

Yet-Another-Markup-Language (YAML)

[YAML Ain't Markup Language \(YAML\)](#) is a data serialization language. It's easy for humans to read, and is commonly used for configuration files.

Guidelines

- Three dashes: start of YAML document
- Specify colon-separated key:value pairs
- Lists: use a dash for each element

Thoughts on human-readable formats

- It's easier to scan than XML and much less verbose; no extra tags.
- Indentation for structure is difficult to parse manually.
- Not as widely supported as other formats :/

```
—
doe: "a deer, a female deer"
ray: "a drop of golden sun"
pi: 3.14159
xmas: true
french-hens: 3
calling-birds:
  - huey
  - dewey
xmas-fifth-day:
  calling-birds: four
  french-hens: 3
  golden-rings: 5
  partridges:
    count: 1
    location: "a pear tree"
  turtle-doves: two
```

JavaScript Object Notation (JSON)

[JavaScript Object Notation \(JSON\)](#) is an open standard data serialization format that's commonly used on the web.

It's based on JavaScript object notation but is language independent. It was standardized in 2013 as ECMA-404.

It has a much simpler syntax compared to XML or YAML.

- Data elements consist of name/value pairs
- Fields are separated by commas
- Curly braces hold objects
- Square brackets hold arrays

JSON is preferred for communications and data transmission.

- It is widely supported by existing programming languages.
- It's the “default choice” for a serializable data format.

JavaScript Object Notation (JSON)

```
{  
  "albums": [  
    {  
      "title": "Empire Burlesque",  
      "artist": "Bob Dylan",  
      "company": "Columbia",  
      "price": "10.90",  
      "year": "1988"  
    },  
    {  
      "title": "Innervision",  
      "artist": "Stevie Wonder",  
      "company": "The Record Plant",  
      "price": "9.90",  
      "year": "1973"  
    }  
  ]  
}
```

Serialization

What is it?

We have objects in memory. We'd like to convert them to one of our data formats to save them. e.g., save a YAML or JSON file.

How can we accomplish this?

[Serialization](#) is a mechanism to convert a data object to a useful format that you can save/stream or otherwise manipulate outside of your program.

- **Serialization**: save your object to a stream.
- **Deserialization**: instantiate an object from your stream.

Let's use this data class in our examples:

```
@Serializable
data class Server(
    val host: String,
    val port: Int,
    val timeoutMs: Long = 5000L,
    val environments: List<String>
)
```

What is it?

Binary Support

kotlin can already stream any object that is serializable i.e., implements that interface. File objects for instance are easily serializable as binary objects.

```
class Emp(var name: String, var id:Int) : Serializable
var file = FileOutputStream("datafile")
var stream = ObjectOutputStream(file) // binary format

var ann = Emp(1001, "Anne Hathaway", "New York")
stream.writeObject(ann) // serialize to a file
```

We're going to focus on human-readable formats as much as we can.

What is it?

JSON Support

To save as JSON, we require a serialization library to handle the conversion. The [kotlinx.serialization](#) library has support for JSON.

It works this way:

- Add the serialization dependency to your `module.yaml` file.
- Use the library to convert data objects directly into JSON strings.
- Manipulate like any other string e.g., save to disk, or stream over the network.
- Deserialization can be used to reverse the process i.e. convert a stream back to an object, as long as you have a corresponding data class.

What is it?

Using kotlin serialization, we can easily convert back and forth between a known data class and a JSON string. Note that this will throw an exception if the conversion fails.

```
{ "host": "localhost",  
  "port": 8080,  
  "environments": [  
    "development",  
    "staging"  
  ] }
```

```
@Test  
fun jsonTest() {  
    val data = Server(host = "localhost", port = 8080,  
        environments = listOf("development", "staging")  
    )  
    val jsonString = Json.encodeToString(data)  
  
    val obj = Json.decodeFromString<Server>(jsonString)  
    assert(obj.host == "localhost")  
    assert(obj.port == 8080)  
    assert(obj.environments == listOf("development", "staging"))  
}
```

What is it?

YAML Support

Kotlin serialization only supports JSON, but third-party libraries support different formats. For example, we can use KAML to serialize YAML files.

```
host: "localhost"
port: 8080
timeoutMs: 5000
environments:
- "development"
- "staging"
```

```
@Test
fun yamlTest() {
    val server = Server(host = "localhost", port = 8080,
        environments = listOf("development", "staging"))
    val yamlString = Yaml.default.encodeToString(server)

    val obj = Yaml.default.decodeFromString<Server>(yamlString)
    assert(server.host == "localhost")
    assert(server.port == 8080)
    assert(server.environments ==
        listOf("development", "staging"))
}
```

What is it?

We'll use a human-readable format e.g., JSON, for storing and transmitting data anywhere that we need it.

Structure + data means that we can process it consistently.

- We can easily convert to/from an object format
 - Easy to work with it! It's just a string.



Don't underestimate the value in being able to read your data in a debugger, or text editor as you're working with it. JSON being human-readable text is one of its biggest advantages.

Bibliography

- [1] M. Richards and N. Ford, *Fundamentals of Software Architecture A Modern Engineering Approach*, 2nd ed. O'Reilly Media Inc., 2009.