
REAL
LIFE
CLEAN ARCHITECTURE
@BattistonMattia

ABOUT ME

Mattia Battiston

@BattistonMattia

mattia.battiston@gmail.com

mattia-battiston

- from Verona, Italy
- software dev & continuous improvement
- Kanban, Lean, Agile “helper”
- Sky Network Services

DISCLAIMER #1

~~MICROSERVICES~~

"Normal" apps

DISCLAIMER #2

~~THE ONE WAY~~

Our experience
(pragmatism, tradeoffs)

COMMON PROBLEMS

Decisions taken too early

Hard to change

Infrequent deploys

Centered around frameworks

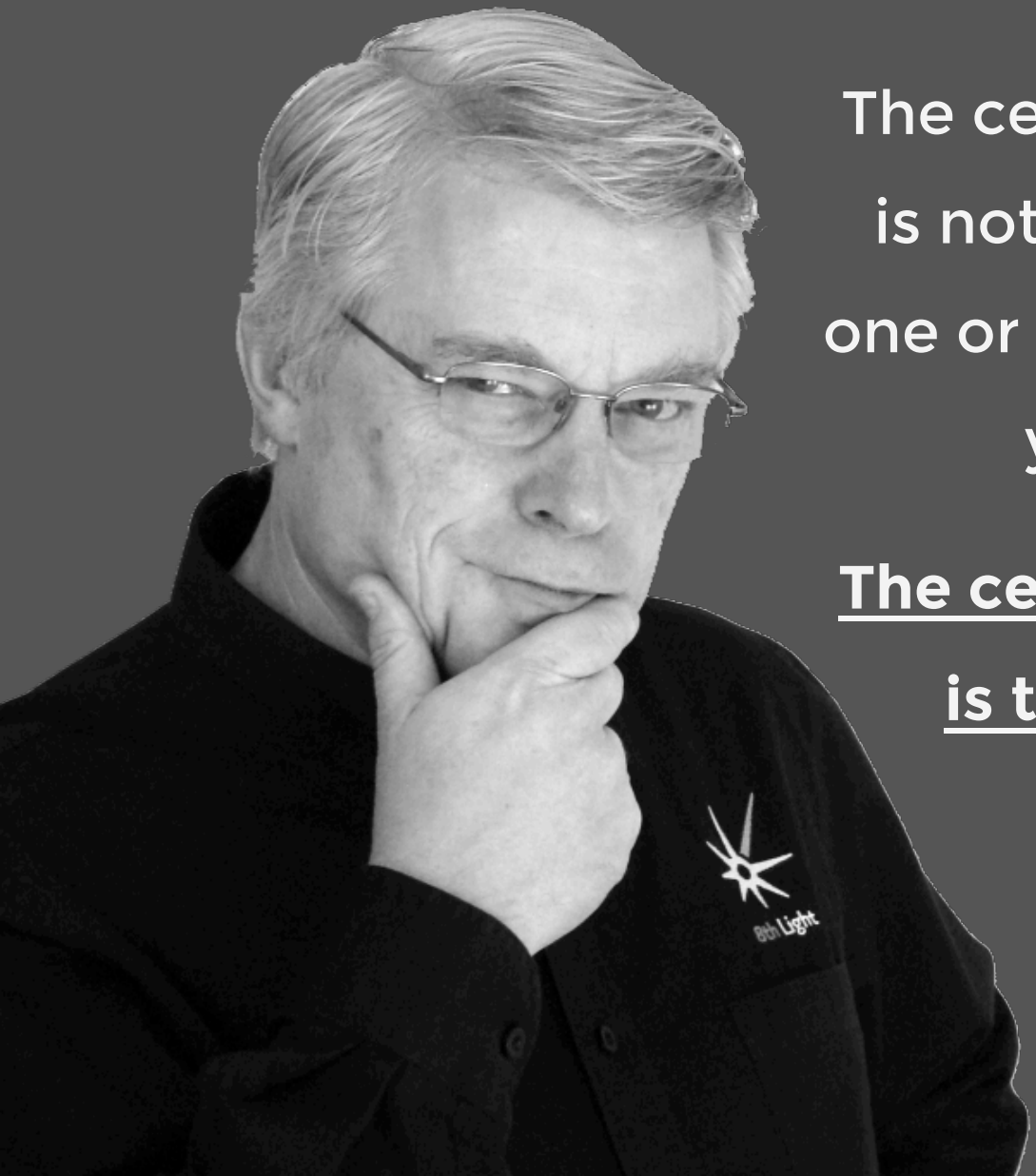
Centered around database

Business logic is spread everywhere

Hard to find things

Focused on technical aspects

Slow, heavy tests



“

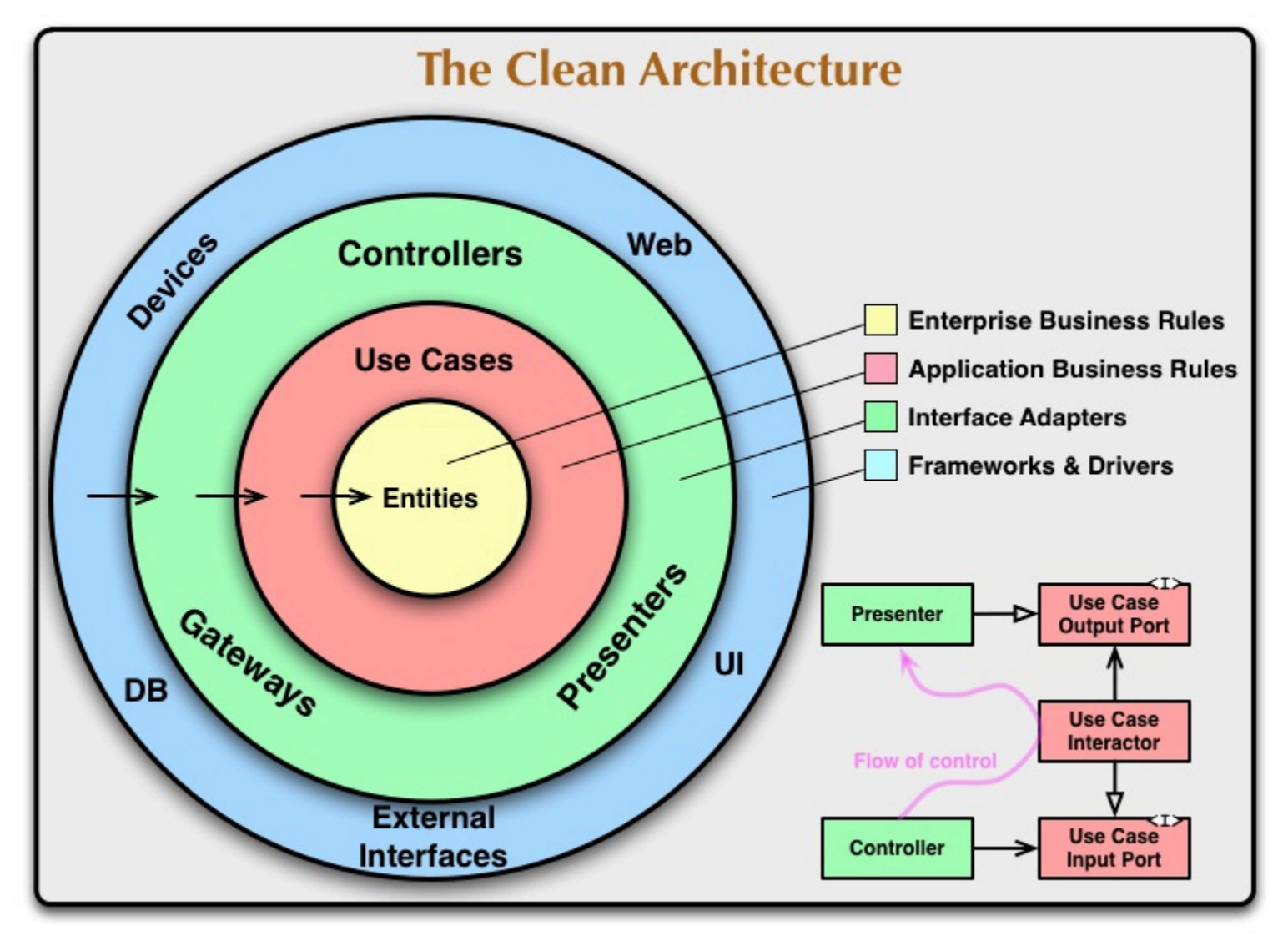
The center of your application is not the database. Nor is it one or more of the frameworks you may be using.

The center of your application is the use cases of your application

Unclebob

Source: NODB

CLEAN ARCHITECTURE



Source: The Clean Architecture

THANK YOU!

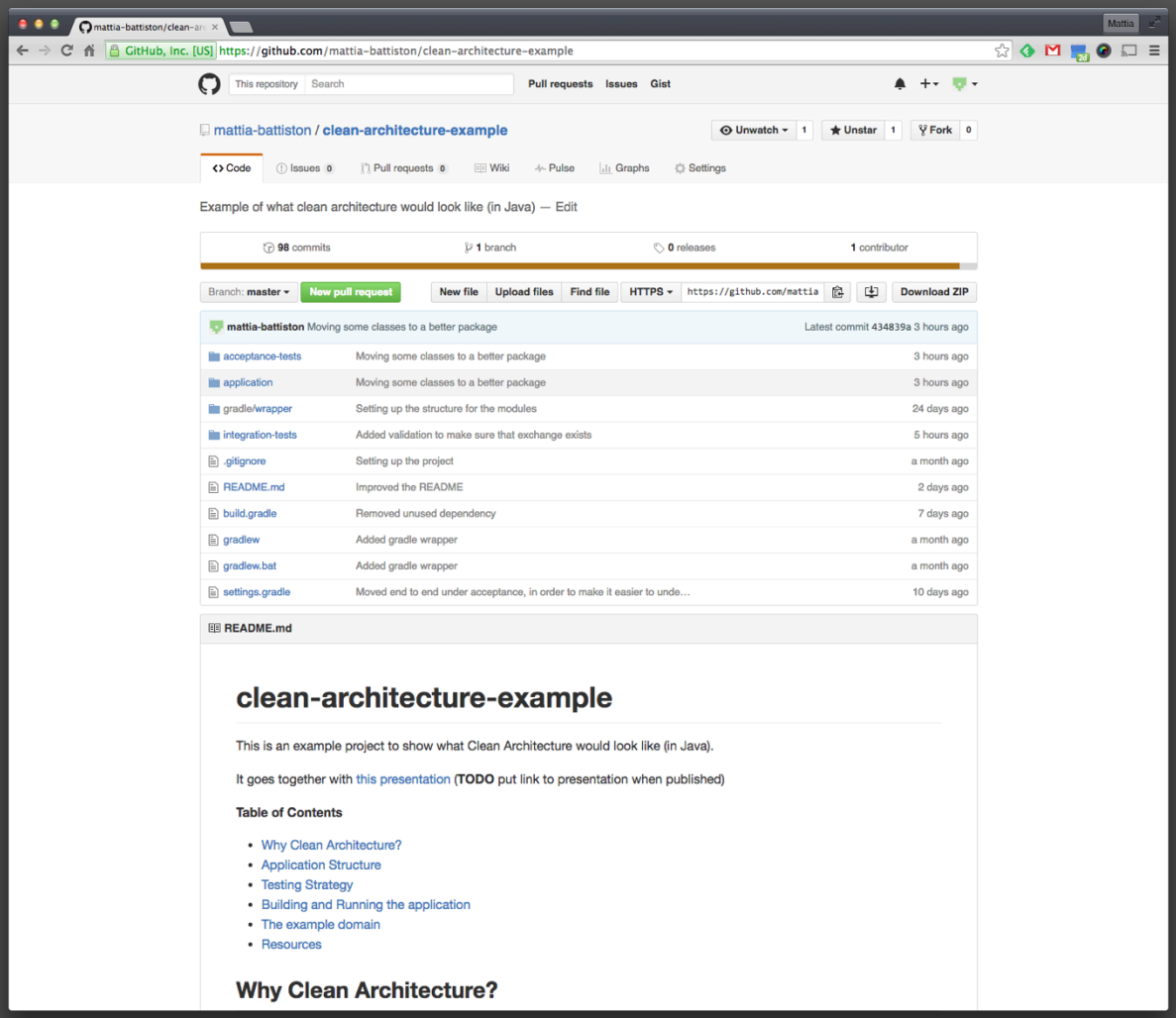
~~THE END~~

Just Kidding,

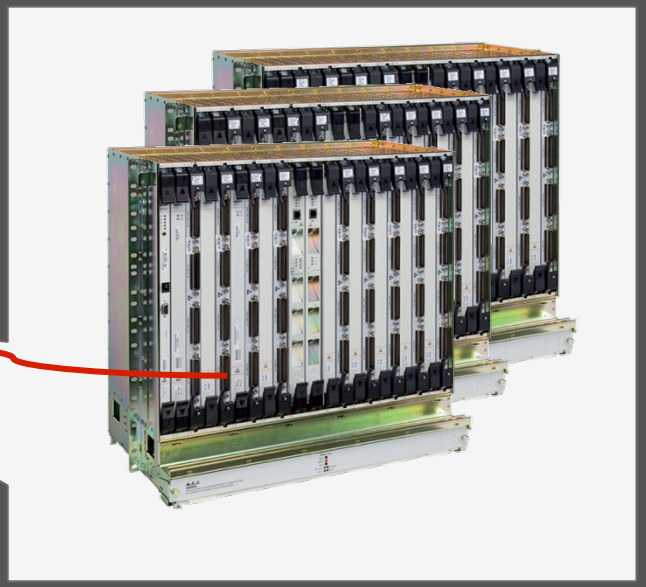
The fun is about to start :-)

EXAMPLE PROJECT

https://github.com/mattia-battiston/clean-architecture-example



EXAMPLE DOMAIN: ISP



Entity:
Telephone Exchange

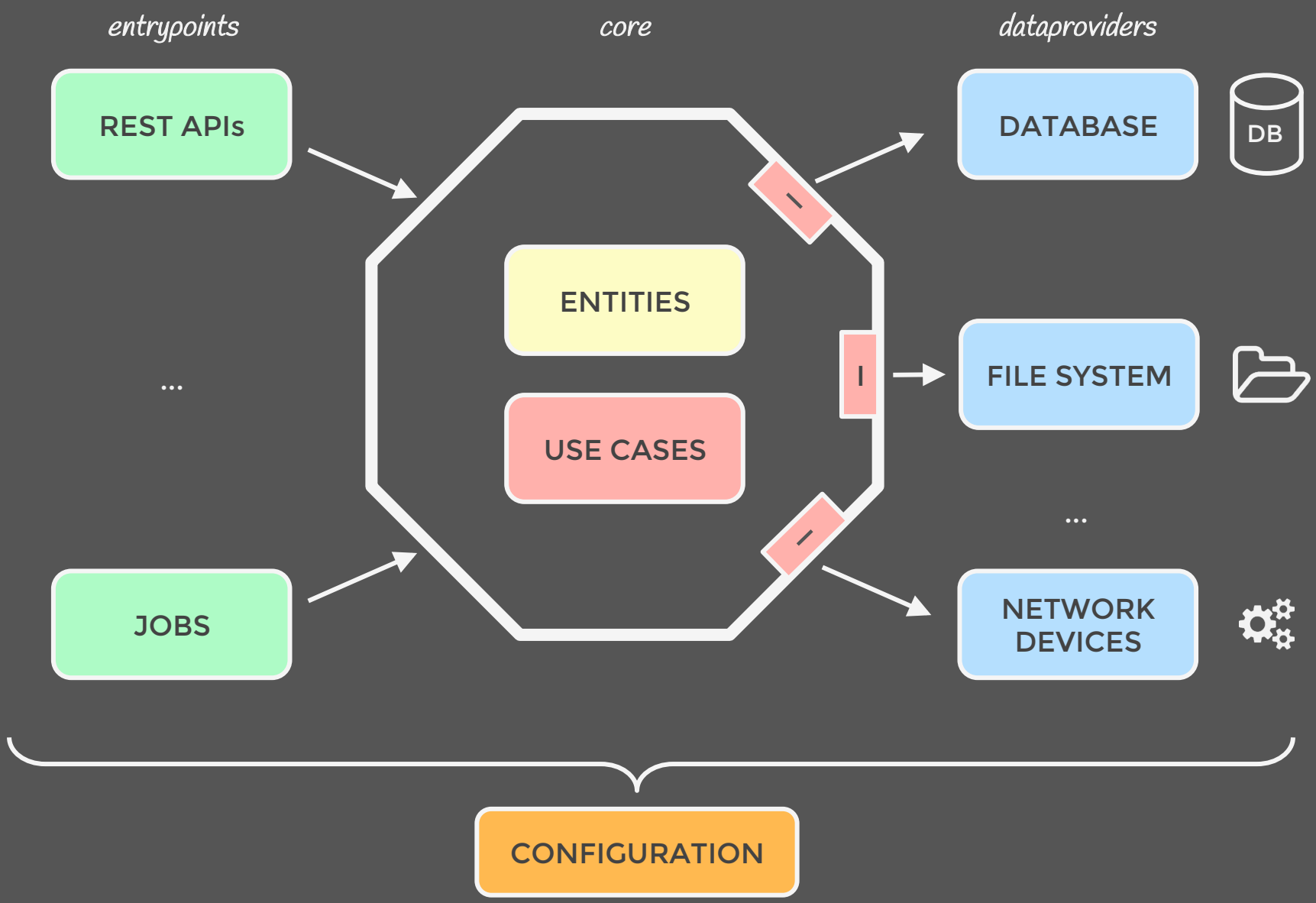
Use case:
Get Capacity
"Have we got space for more customers? Can we sell more broadband?"

Entity:
Broadband Access Device

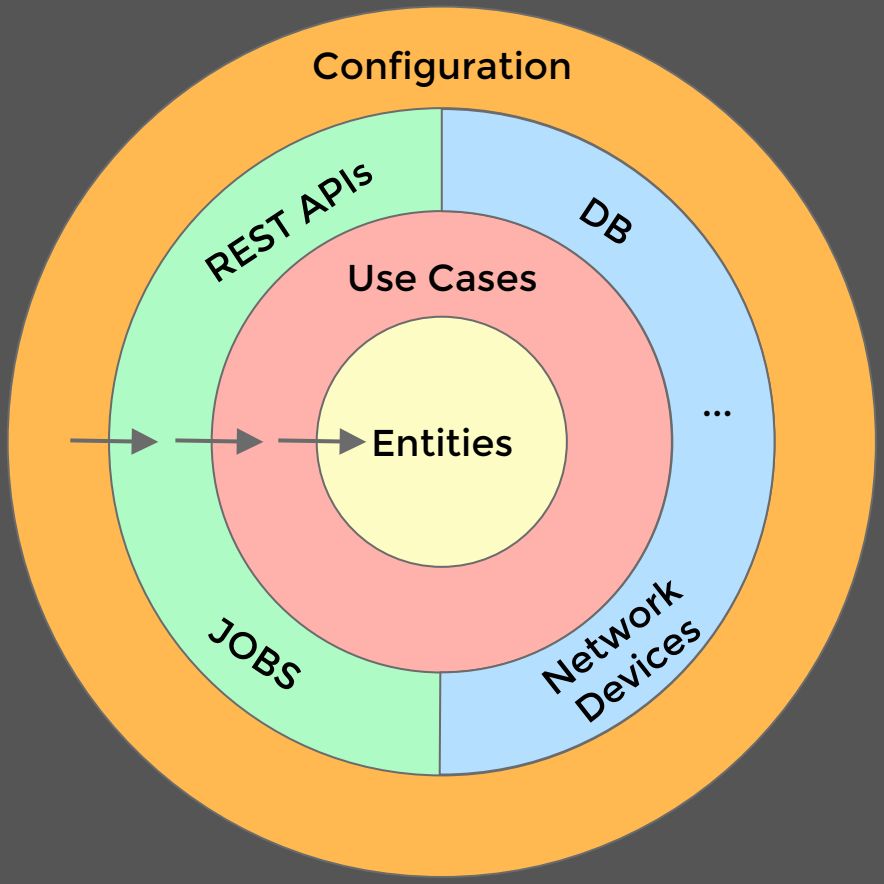
Use case:
Reconcile
"Has any physical device changed? (e.g. new serial number)"

Use case:
Get Details
"Details of a particular device?"

OUR CLEAN ARCHITECTURE



A.k.a (unclebob's style)



\+ Core: business logic

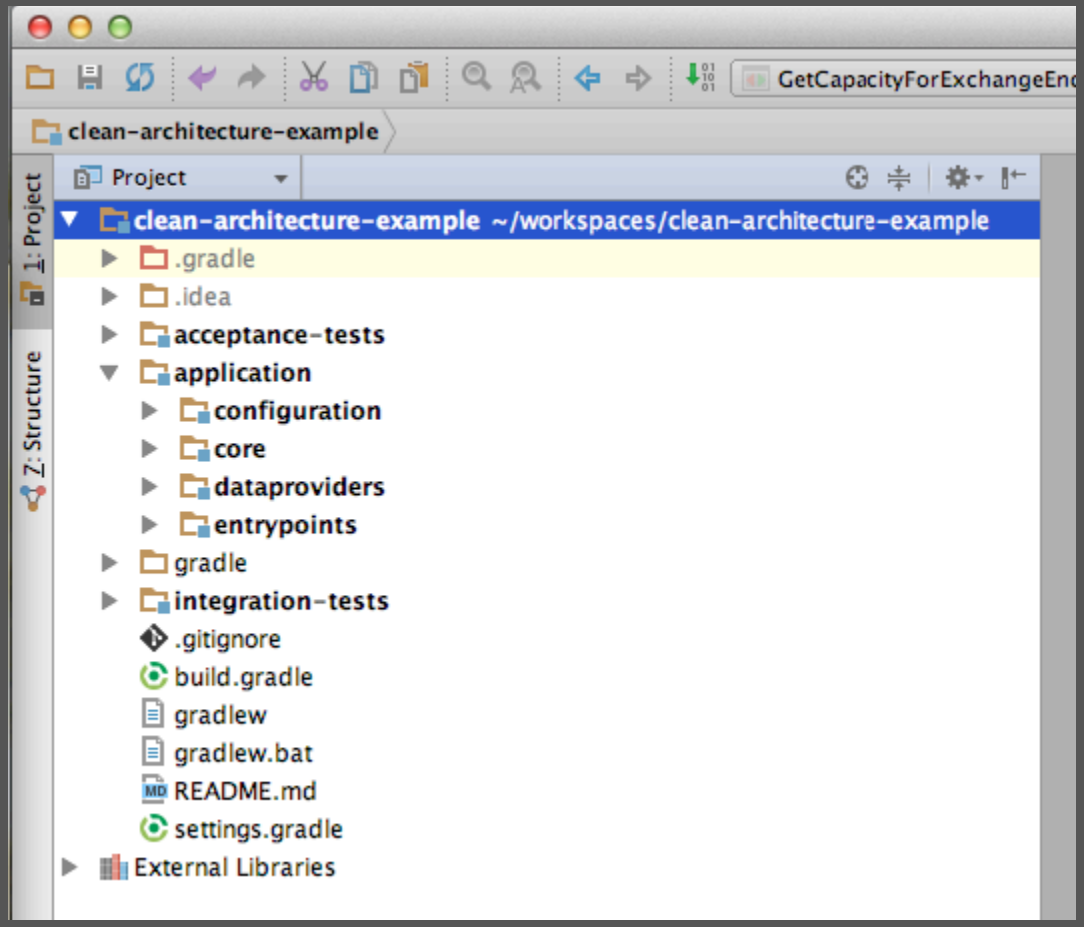
Entrypoints: access to the application

Data Providers: retrieve and store information

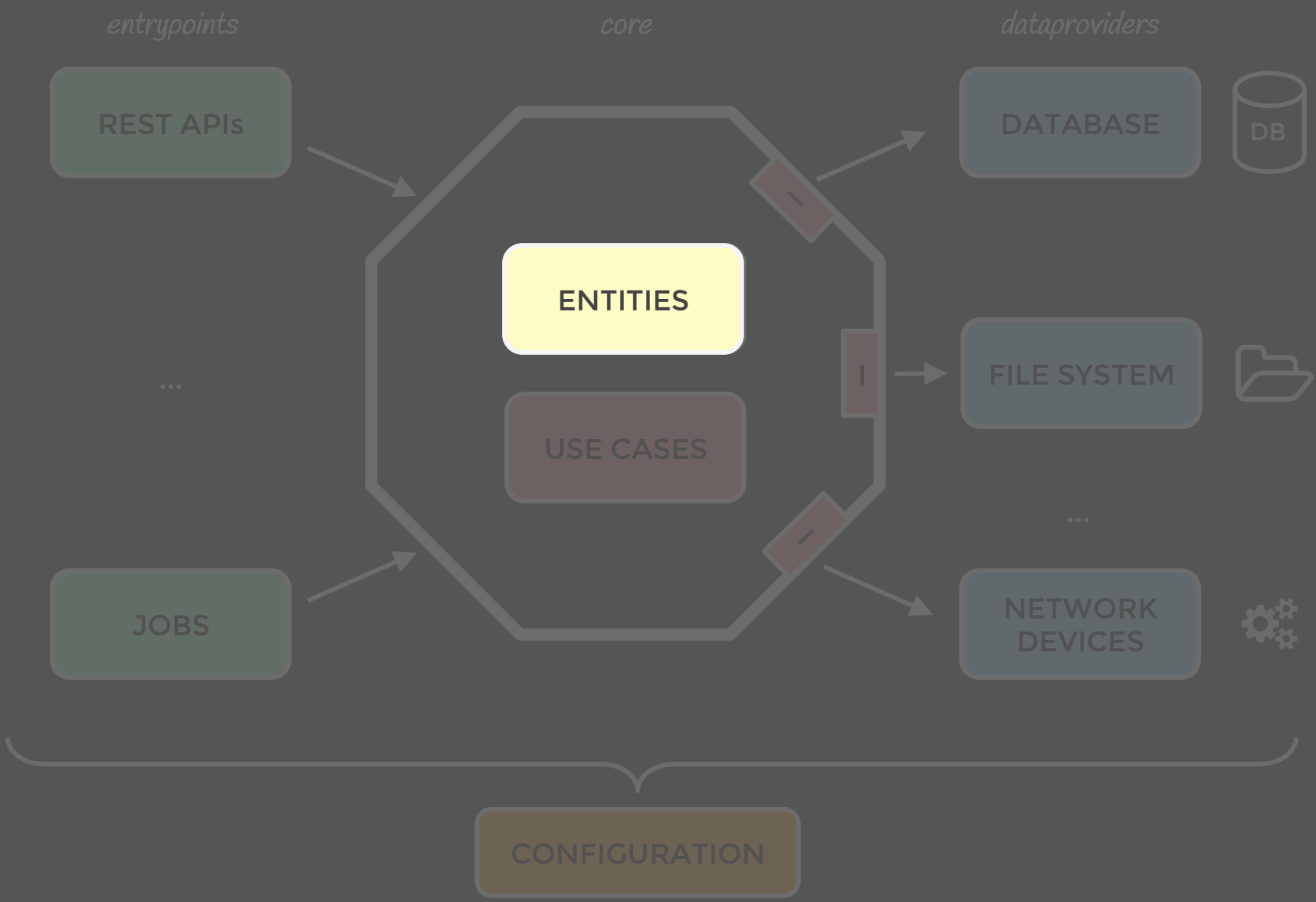
Configuration: wires everything together

** arrows represent dependency*

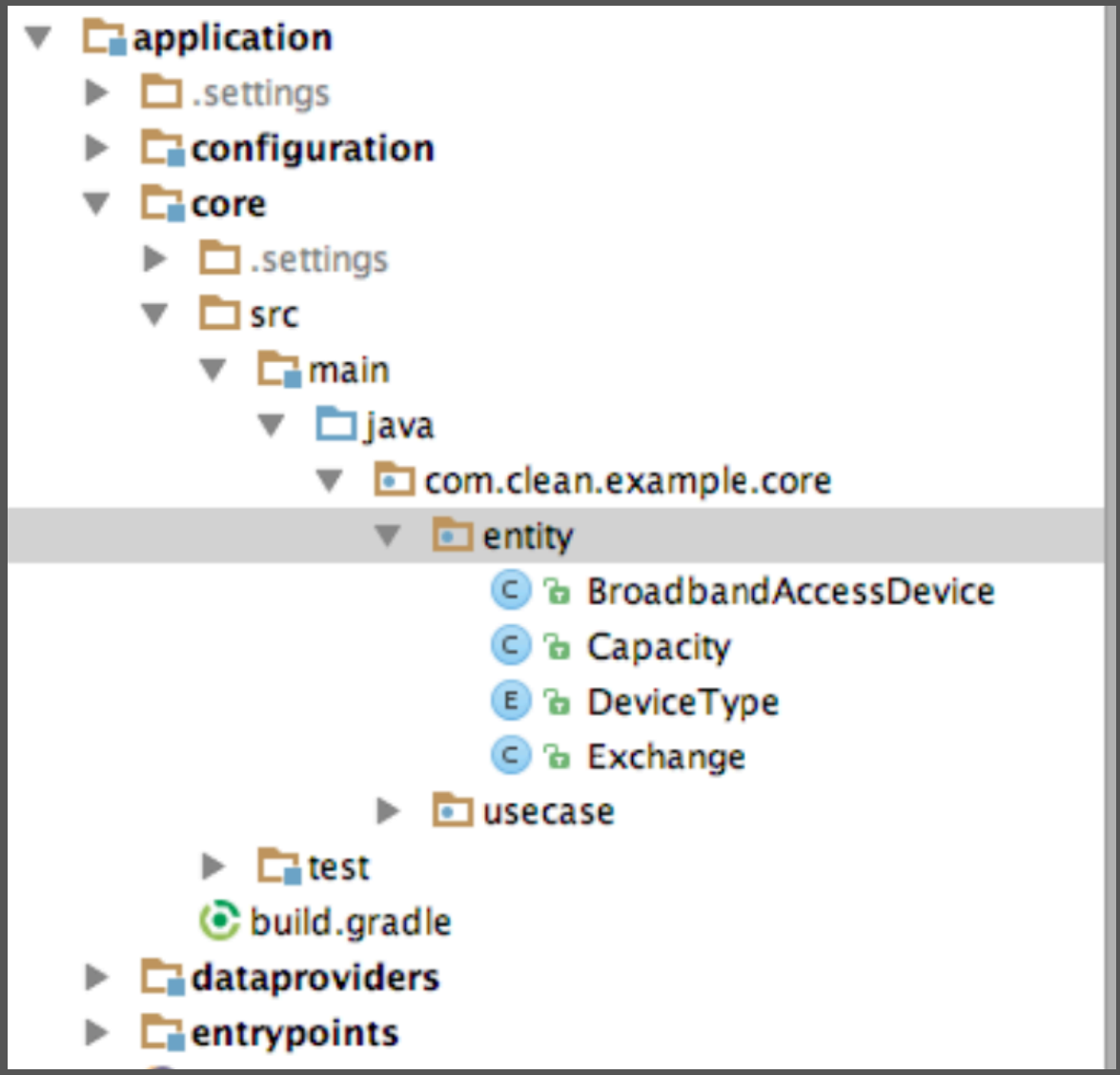
PROJECT STRUCTURE



ENTITY


entrypoints
core
dataproviders
REST APIs
...
JOBS
ENTITIES
USE CASES
I
DATABASE
DB
FILE SYSTEM
...
NETWORK DEVICES
CONFIGURATION

ENTITY


application
.settings
configuration
core
.settings
src
main
java
com.clean.example.core
entity
BroadbandAccessDevice
Capacity
DeviceType
Exchange
usecase
test
build.gradle
dataproviders
entrypoints

ENTITY

```
public class Exchange {

    private final String code;
    private final String name;
    private final String postCode;

    public Exchange(String code, String name, String postCode) {
        this.code = code;
        this.name = name;
        this.postCode = postCode;
    }

    // … getters

    @Override
    public String toString() {
        return ...;
    }

}
```

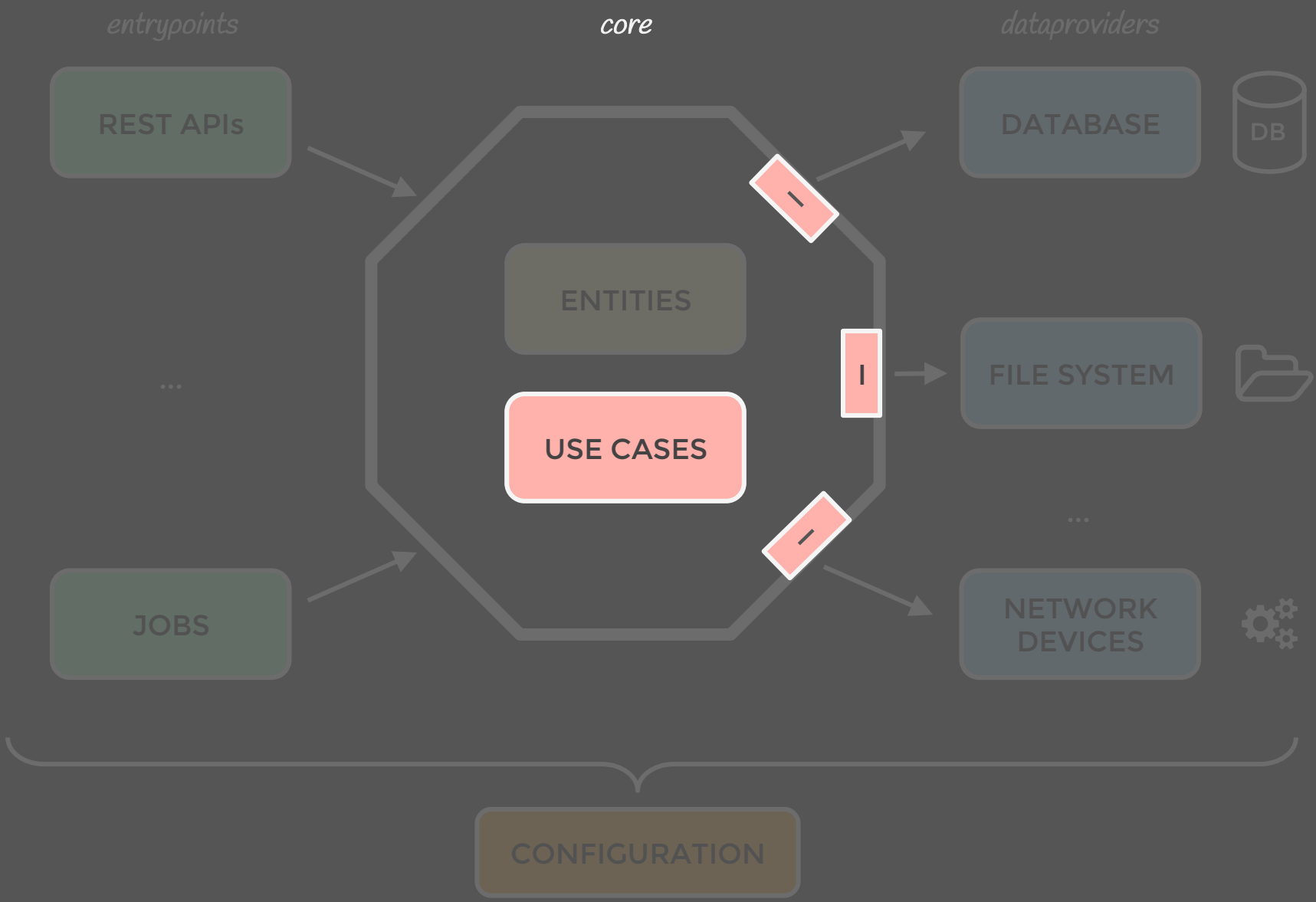
ENTITY

```
public class BroadbandAccessDevice {

    private Exchange exchange;

    private String hostname;
    private String serialNumber;
    private DeviceType type;
    private int availablePorts;

    public BroadbandAccessDevice(String hostname, ...) {
        this.hostname = hostname;
        this.serialNumber = serialNumber;
        this.type = type;
    }

    // … (getters, some setters)

    @Override
    public String toString() {
        return ...;
    }
}
```

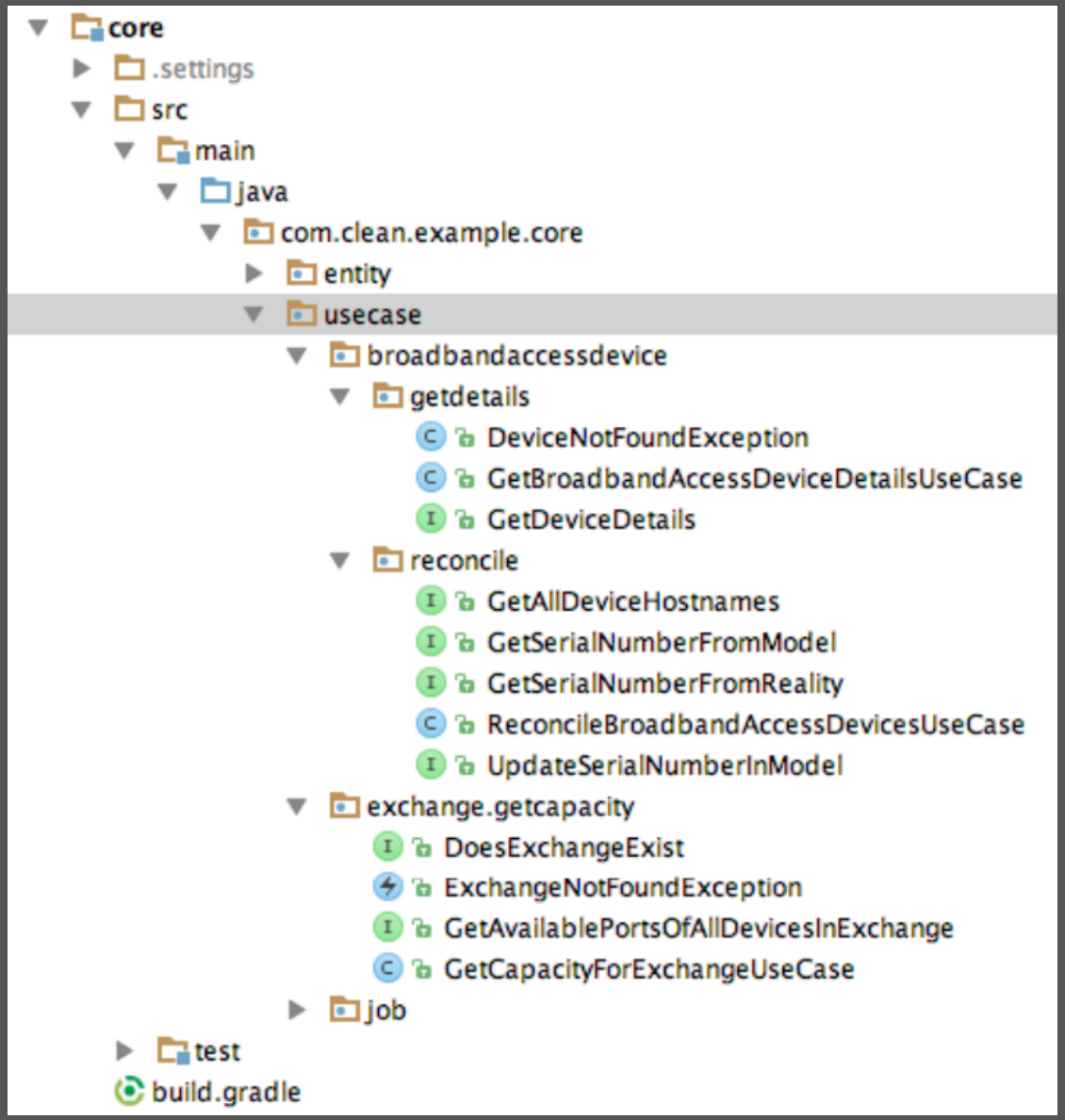
ENTITY

- Domain objects
  e.g. “Exchange”, “Broadband Access Device”
- Entity-specific business rules
  e.g. hostname format
- Tiny Types are useful
  e.g. Hostname instead of String
- No Frameworks

USE CASE


entrypoints
core
dataproviders
REST APIs
...
JOBS
ENTITIES
USE CASES
I
I
I
DATABASE
DB
FILE SYSTEM
...
NETWORK DEVICES
CONFIGURATION

USE CASE


core
.settings
src
main
java
com.clean.example.core
entity
usecase
broadbandaccessdevice
getdetails
DeviceNotFoundException
GetBroadbandAccessDeviceDetailsUseCase
GetDeviceDetails
reconcile
GetAllDeviceHostnames
GetSerialNumberFromModel
GetSerialNumberFromReality
ReconcileBroadbandAccessDevicesUseCase
UpdateSerialNumberInModel
exchange.getcapacity
DoesExchangeExist
ExchangeNotFoundException
GetAvailablePortsOfAllDevicesInExchange
GetCapacityForExchangeUseCase
job
test
build.gradle

USE CASE

```
public class GetCapacityForExchangeUseCase {

    private DoesExchangeExist dataProvider1;
    private GetAvailablePortsOfAllDevicesInExchange dataProvider2;

    // …

    public Capacity getCapacity(String exchangeCode) {
        failIfExchangeDoesNotExist(exchangeCode);
        List<BroadbandAccessDevice> devices =
             dataProvider2
             .getAvailablePortsOfAllDevicesInExchange(exchangeCode);
        boolean hasAdslCapacity = hasCapacityFor(devices, DeviceType.ADSL);
        boolean hasFibreCapacity = hasCapacityFor(devices, DeviceType.FIBRE);
        return new Capacity(hasAdslCapacity, hasFibreCapacity);
    }

    // …

    private void failIfExchangeDoesNotExist(String exchangeCode) {
        boolean exchangeExists = dataProvider1.doesExchangeExist(exchangeCode);
        if(!exchangeExists) throw new ExchangeNotFoundException();
    }
}
```

USE CASE

```
public interface DoesExchangeExist {
   boolean doesExchangeExist(String exchange);
}
```

—

```
public interface GetAvailablePortsOfAllDevicesInExchange {
   List<BroadbandAccessDevice>
         getAvailablePortsOfAllDevicesInExchange(String exchangeCode);
}
```

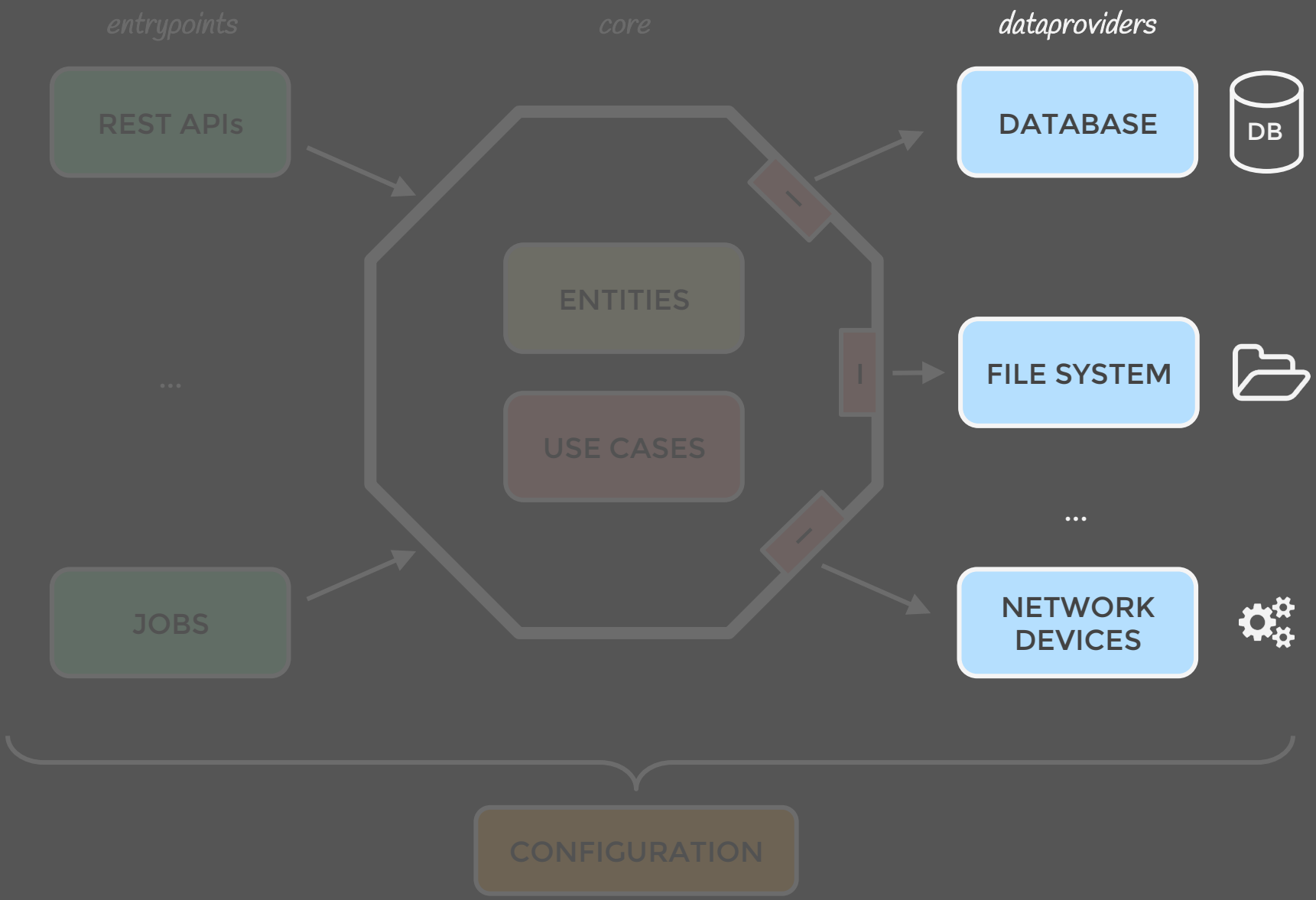
—

```
public class ExchangeNotFoundException extends RuntimeException {
}
```

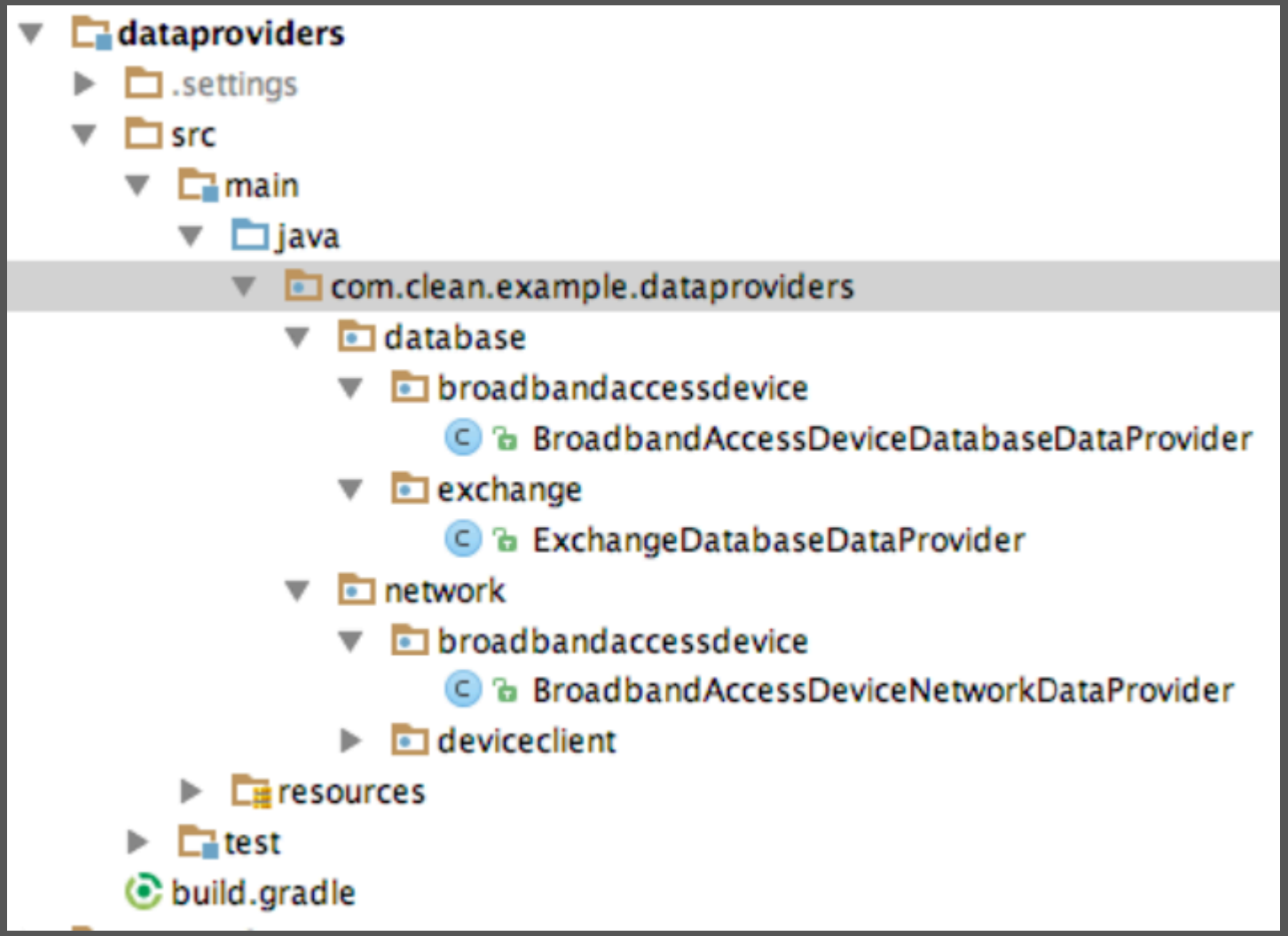
USE CASE

- Pure business logic
  e.g. algorithm to calculate capacity
- Defines *Interfaces* for the data that it requires
  e.g. getAvailablePorts
- Uses entities and dataproviders
- Throws business-specific exceptions
- Not affected by changes in database or presentation
- Plain Java (no frameworks)
- We like single-method interfaces
  A.k.a. Interface Segregation

DATA PROVIDER



DATA PROVIDER



DATA PROVIDER

```
public class BroadbandAccessDeviceDatabaseDataProvider
    implements GetAllDeviceHostnames, GetSerialNumberFromModel, ... {

   private JdbcTemplate jdbcTemplate;

   @Override
   public List<String> getAllDeviceHostnames() {
       return jdbcTemplate.queryForList(
        "SELECT hostname FROM clean_architecture.bb_access_device",
        String.class);
   }

   @Override
   public String getSerialNumber(String hostname) {
       try {
           return jdbcTemplate.queryForObject(
                   "SELECT serial_number FROM
         clean_architecture.bb_access_device
             WHERE hostname = ?",
                   String.class, hostname);
       } catch (IncorrectResultSizeDataAccessException e) {
           return null;
       }
   }

   // …
```

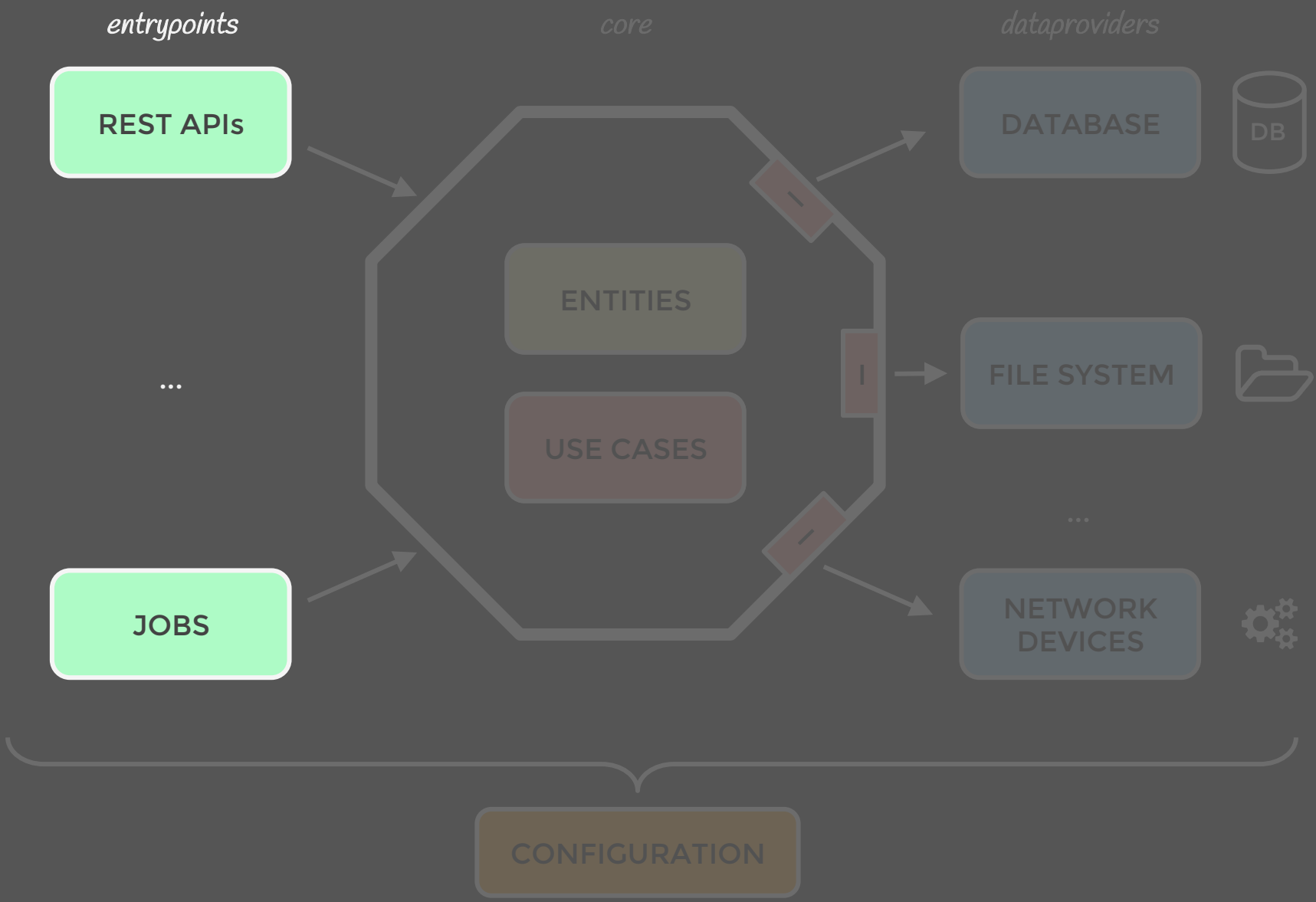
DATA PROVIDER

```
public class BroadbandAccessDeviceNetworkDataProvider
                implements GetSerialNumberFromReality {

    private DeviceClient deviceClient;

    // …

    @Override
    public String getSerialNumber(String hostname) {
        try {
            return deviceClient.getSerialNumber(hostname);
        } catch (DeviceConnectionTimeoutException e) {
            return null;
        }
    }

}
```

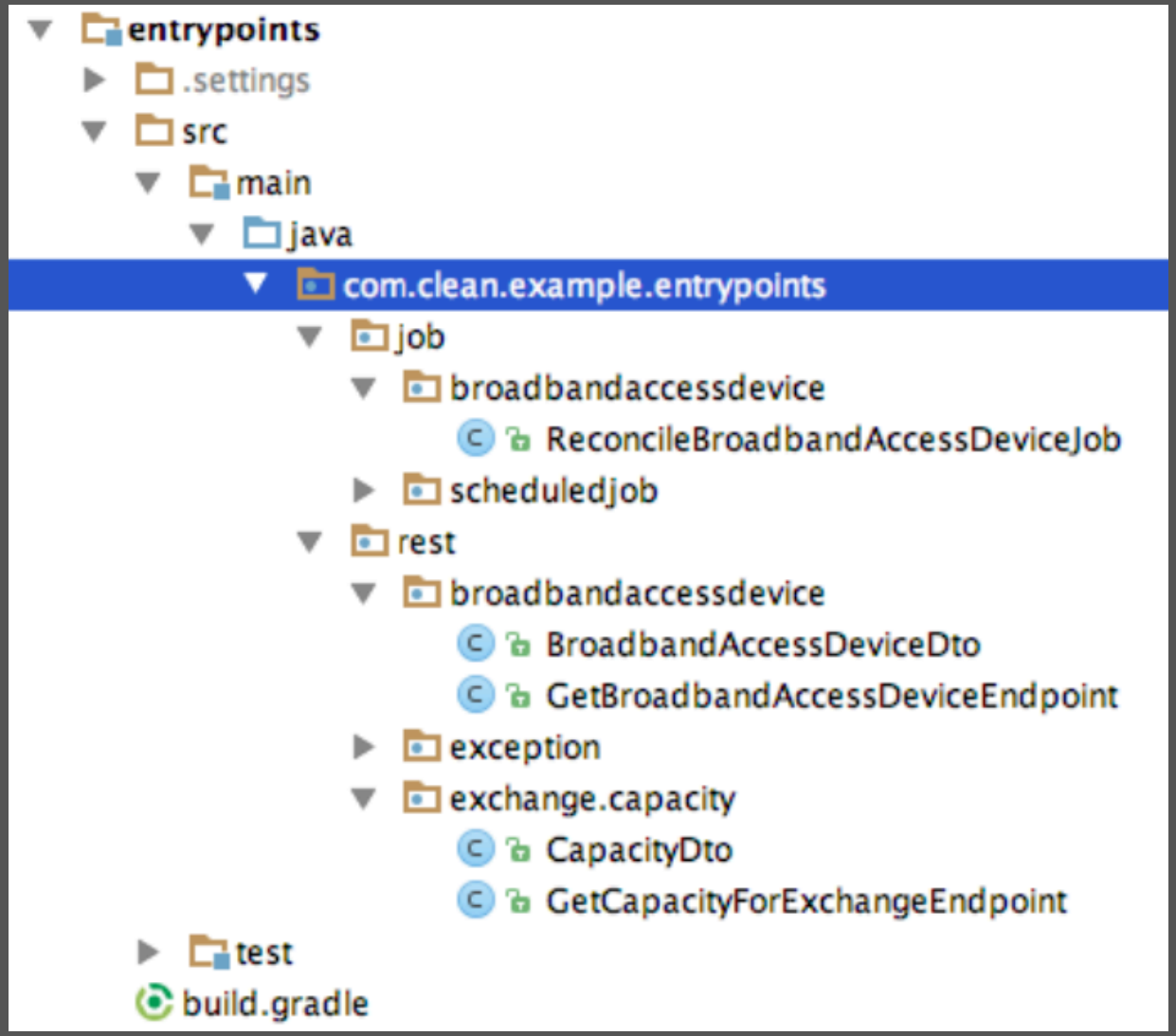
DATA PROVIDER

- Implements *Interfaces* defined by use case
  e.g. retrieving serial number of a device
- Hides all details about where the data is from
- Could have multiple implementations
  e.g. from DB or from File System
- Uses whatever framework/library is appropriate
  e.g. spring-jdbc, hibernate, snmp4j, etc.
- If using ORM, you'd have a new set of entities
  A.k.a. decoupled from business entities

ENTRYPOINT


entrypoints
core
dataproviders
REST APIs
...
JOBS
ENTITIES
USE CASES
I
DATABASE
DB
FILE SYSTEM
...
NETWORK DEVICES
CONFIGURATION

ENTRYPOINT


entrypoints
.settings
src
main
java
com.clean.example.entrypoints
job
broadbandaccessdevice
ReconcileBroadbandAccessDeviceJob
scheduledjob
rest
broadbandaccessdevice
BroadbandAccessDeviceDto
GetBroadbandAccessDeviceEndpoint
exception
exchange.capacity
CapacityDto
GetCapacityForExchangeEndpoint
test
build.gradle

ENTRYPOINT

```
@RestController
public class GetCapacityForExchangeEndpoint {

    public static final String API_PATH = "/exchange/{exchangeCode}/capacity";

    private GetCapacityForExchangeUseCase getCapacityForExchangeUseCase;

    // …

    @RequestMapping(value = API_PATH, method = GET)
    public CapacityDto getCapacity(@PathVariable String exchangeCode) {
        try {
            Capacity capacity =
                  getCapacityForExchangeUseCase.getCapacity(exchangeCode);
            return toDto(capacity);
        } catch (ExchangeNotFoundException e) {
            LOGGER.info("Exchange not found: {}", exchangeCode);
            throw new NotFoundException();
        }
    }

    private CapacityDto toDto(Capacity capacity) {
        return new CapacityDto(
             capacity.hasAdslCapacity(), capacity.hasFibreCapacity());
    }
}
```

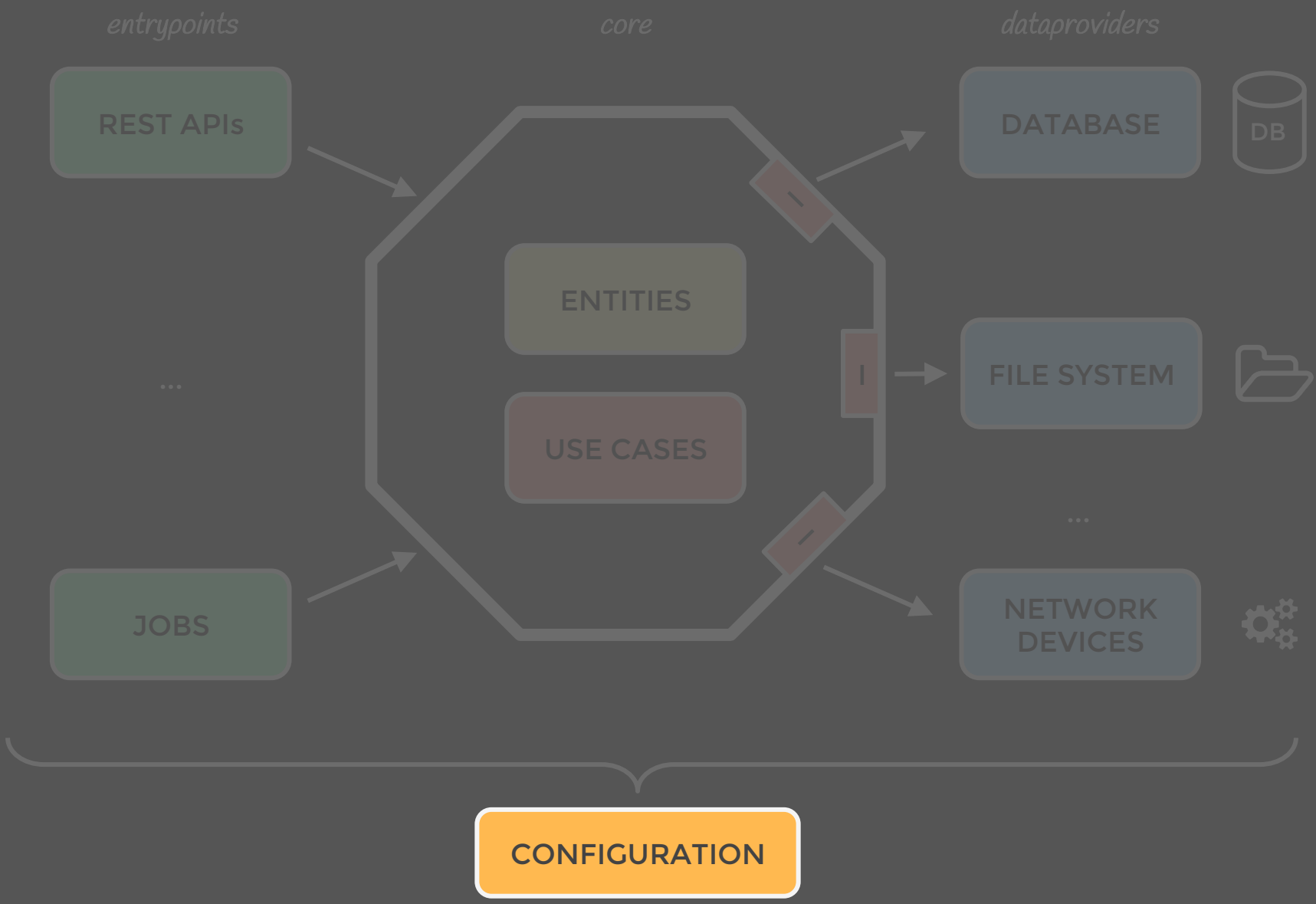
ENTRYPOINT

```
public class ReconcileBroadbandAccessDeviceJob implements ScheduledJob {

    private ReconcileBroadbandAccessDevicesUseCase useCase;
    private final JobResults jobResults;

    // …

    @Override public String getName() { return "..."; }

    @Override public long getInitialDelay() { return 0; }

    @Override public long getPeriod() { return 5; }

    @Override public TimeUnit getTimeUnit() { return TimeUnit.SECONDS; }

    @Override
    public void run() {
        LOGGER.info("Job Starting: {}...", getName());
        JobResultsCount jobResultsCount = jobResults.createJobResultsCount();
        OnSuccess success = jobResultsCount::success;
        OnFailure failure = jobResultsCount::failure;
        reconcileBroadbandAccessDevicesUseCase.reconcile(success, failure);
        jobResults.recordJobResults(this, jobResultsCount);
        LOGGER.info("Job Completed: {}", getName());
    }
}
```

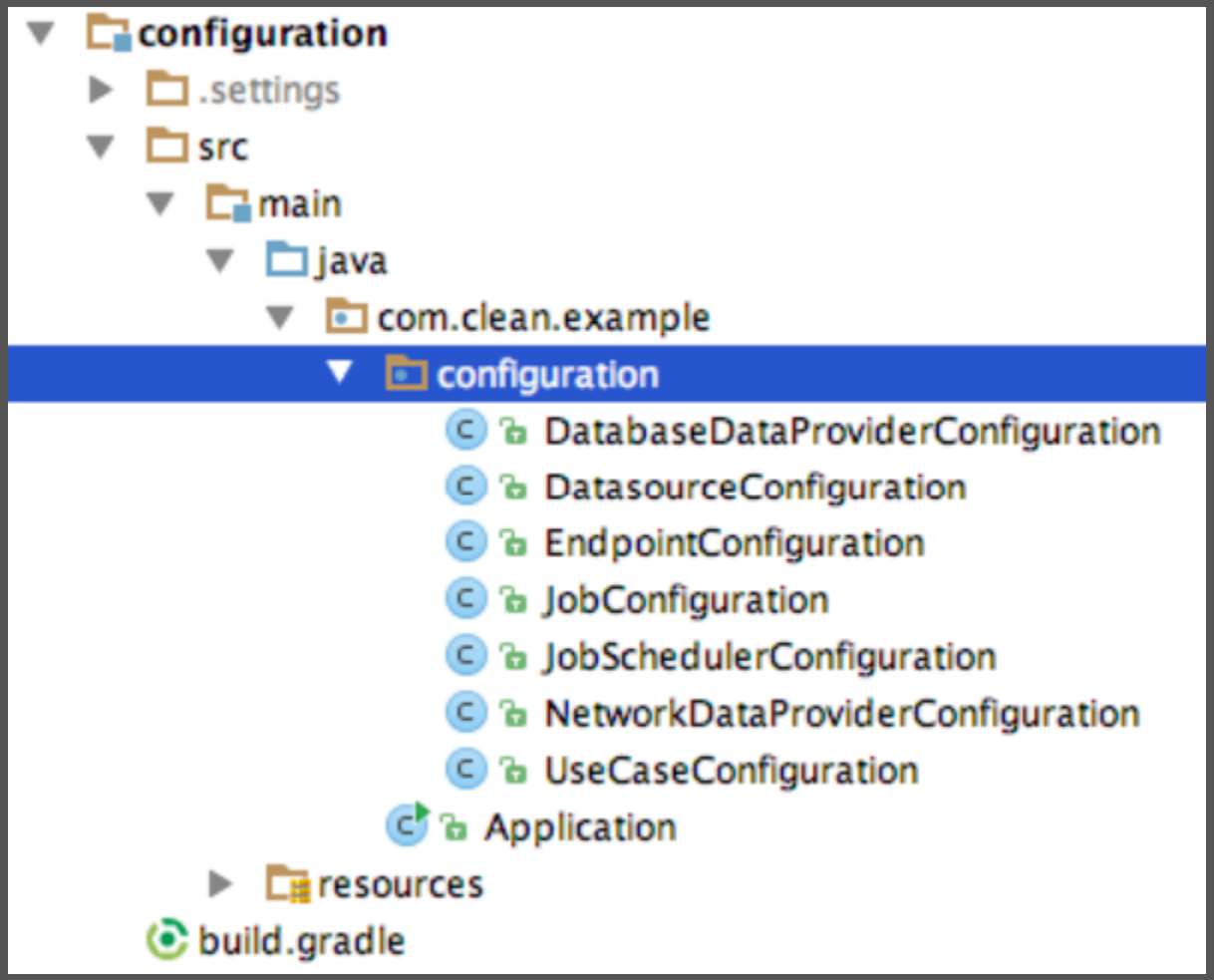
ENTRYPOINT

- Fires up a use case
- Has no business logic
- Has conversion logic
  e.g. uses DTOs to return result to WEB
- Hides all details about delivery mechanism
- GUI would be an entrypoint
  e.g. controllers would start use cases
- Uses whatever framework/library is appropriate
  e.g. spring-mvc, jersey, quartz, etc.

CONFIGURATION



CONFIGURATION



CONFIGURATION

```
@Configuration
@EnableAutoConfiguration
@ComponentScan(basePackages = "com.clean.example.configuration")
public class Application {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

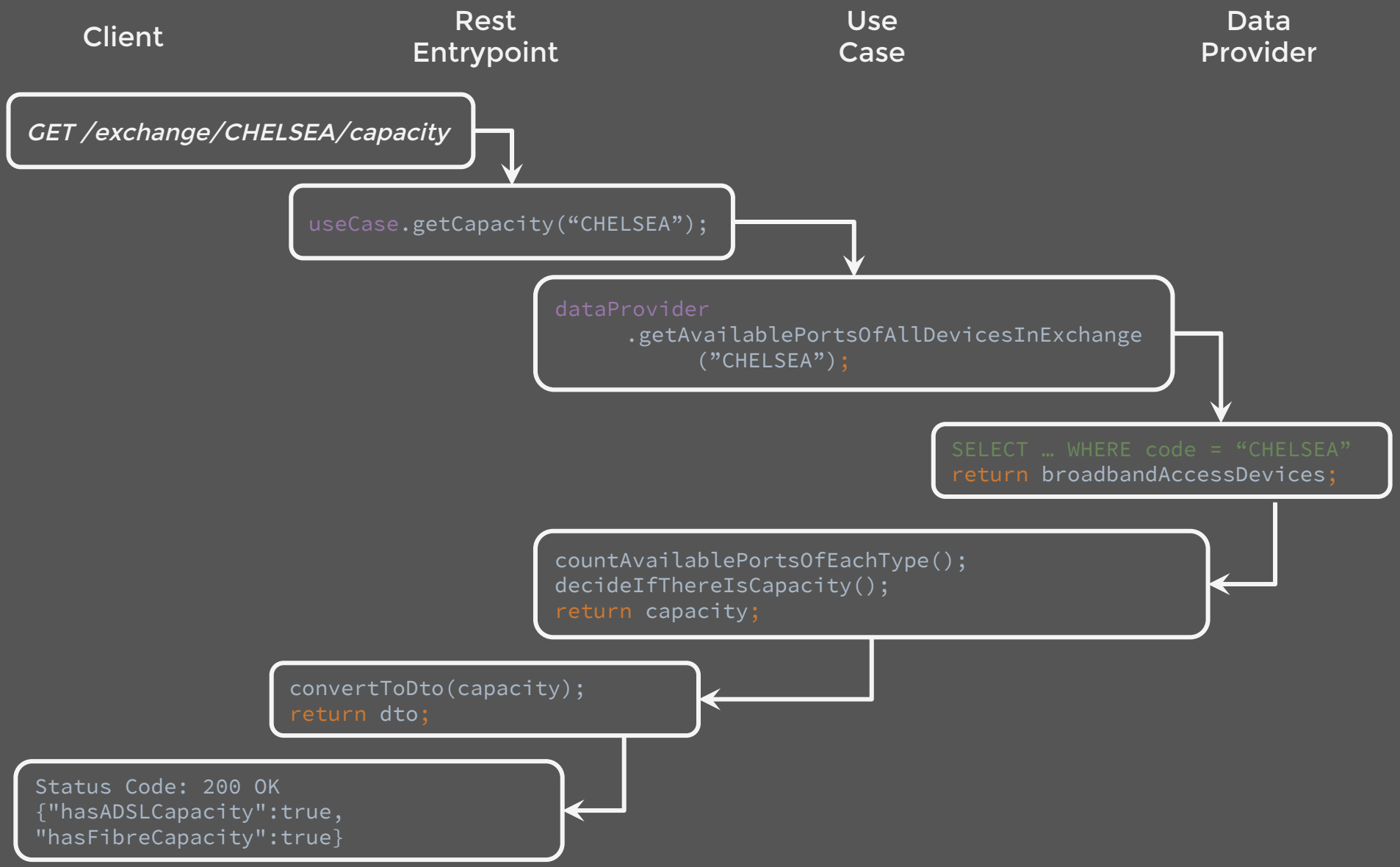
```
@Configuration
public class EndpointConfiguration {
    @Bean
    public GetBroadbandAccessDeviceEndpoint getBroadbandAccessDeviceEndpoint() {
        return new GetBroadbandAccessDeviceEndpoint(...);
    }
    // …other endpoints…
}
```

```
@Configuration
public class WebServerConfiguration {
    // …wires web server with framework…
}
```

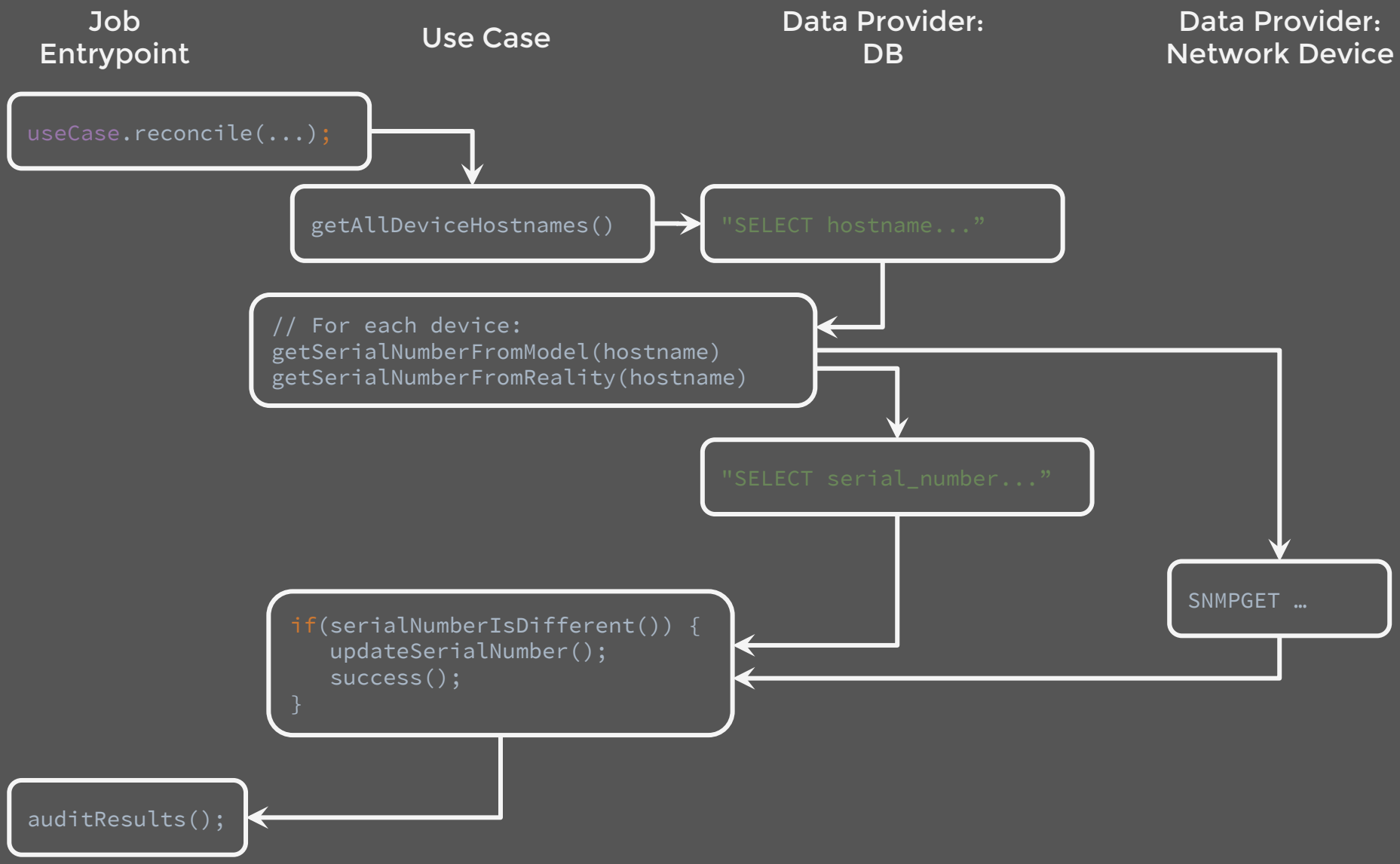
CONFIGURATION

- Wires everything together
  e.g. using Spring or simply initialising objects
- Isolates and hides frameworks
  e.g. Spring is only here
- Has the “dirty” details to make things work
  e.g. initialise datasource, setup web server, etc.

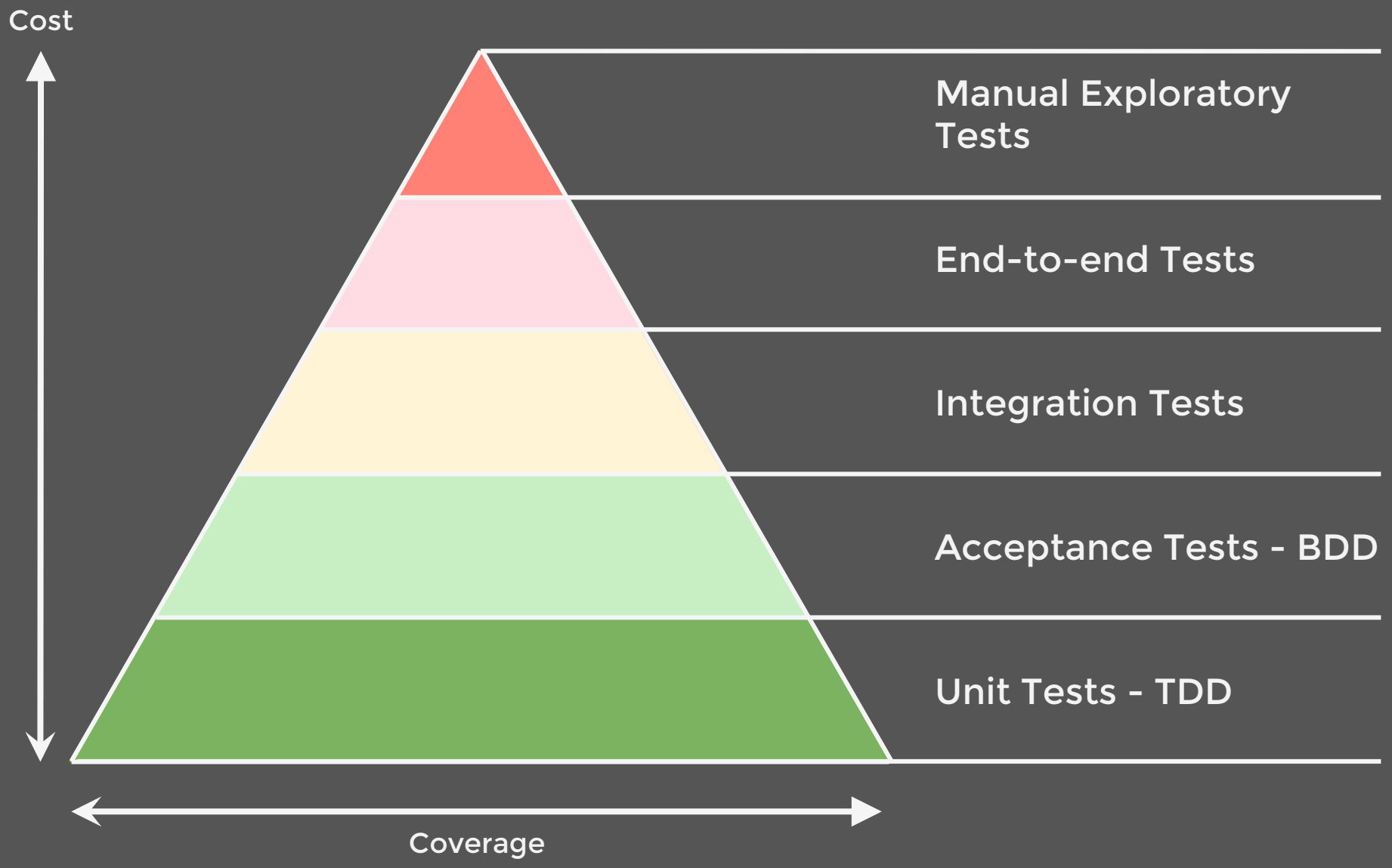
EXAMPLE: GET CAPACITY



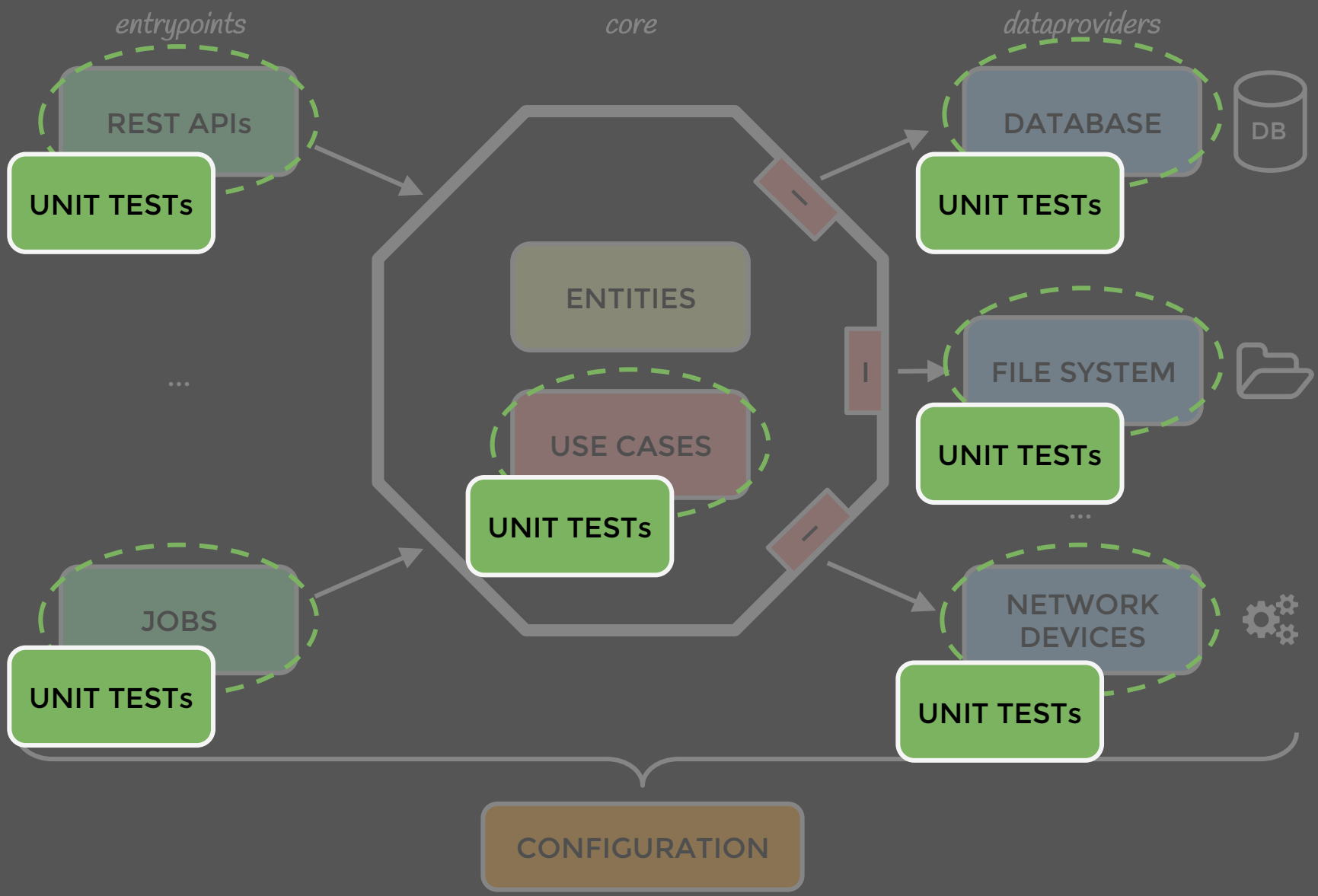
EXAMPLE: RECONCILE DEVICES



TESTING STRATEGY



UNIT TESTS


entrypoints
core
dataproviders
REST APIs
UNIT TESTs
DATABASE
DB
UNIT TESTs
ENTITIES
...
I
FILE SYSTEM
USE CASES
UNIT TESTs
UNIT TESTs
...
JOBS
NETWORK DEVICES
UNIT TESTs
UNIT TESTs
CONFIGURATION

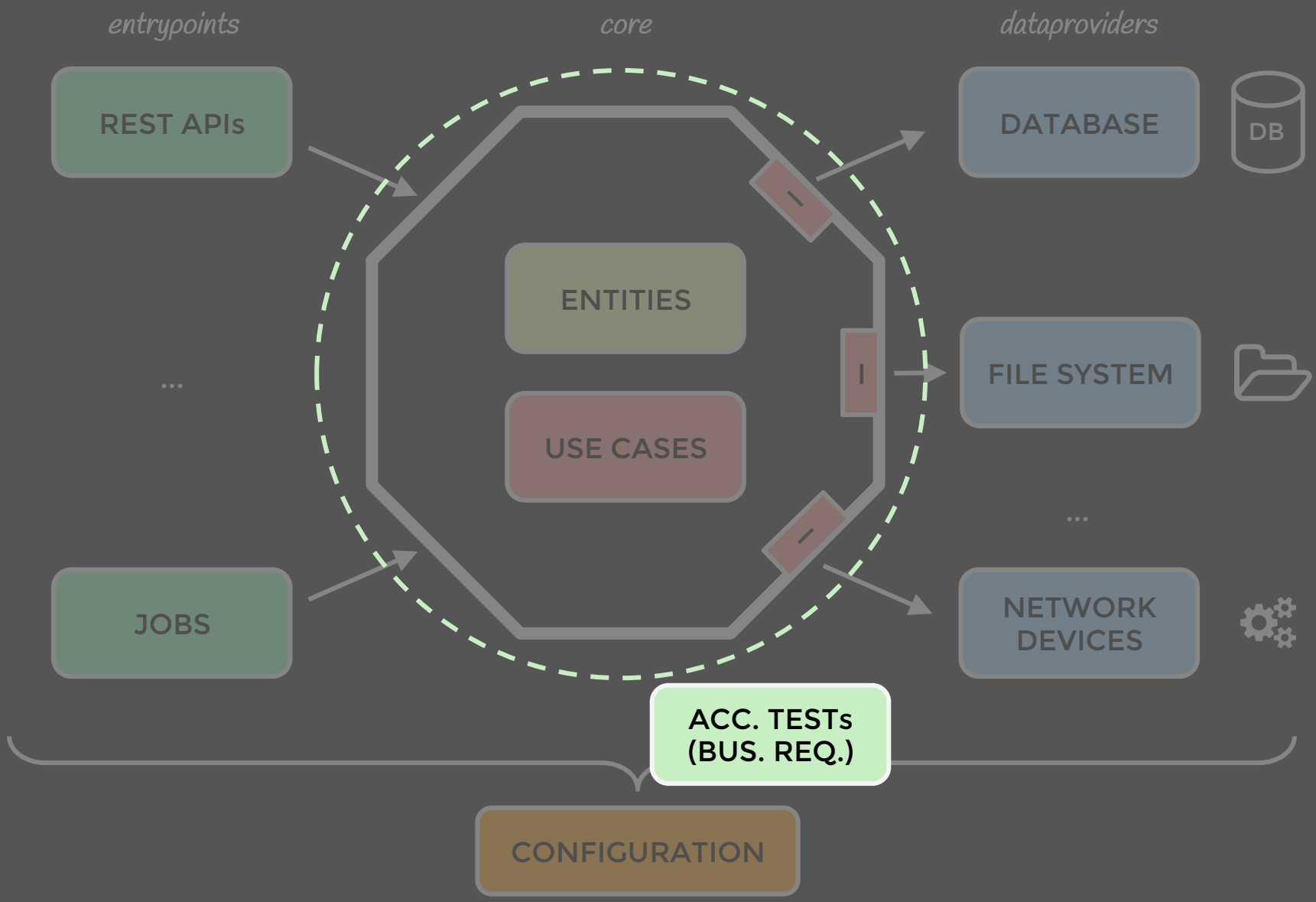
UNIT TESTS

```
// …

GetCapacityForExchangeEndpoint endpoint = // class under test
GetCapacityForExchangeUseCase useCase = // mock

@Test
public void returnsTheCapacityForAnExchange() throws Exception {
   givenThereIsCapacityForAnExchange();

   CapacityDto capacity = endpoint.getCapacity(EXCHANGE_CODE);

   assertThat(capacity.getHasADSLCapacity()).isTrue();
   assertThat(capacity.getHasFibreCapacity()).isTrue();
}

private void givenThereIsCapacityForAnExchange() {
   when(useCase.getCapacity(EXCHANGE_CODE))
        .thenReturn(new Capacity(true, true));
}

// …
```

UNIT TESTS

- TDD
  A.k.a. Tests first, to drive design
- Cover every little detail
  A.k.a. Aim for 100% coverage
- “Dev to dev” documentation
  A.k.a. What should this class do?
- Test individual classes in isolation, very fast

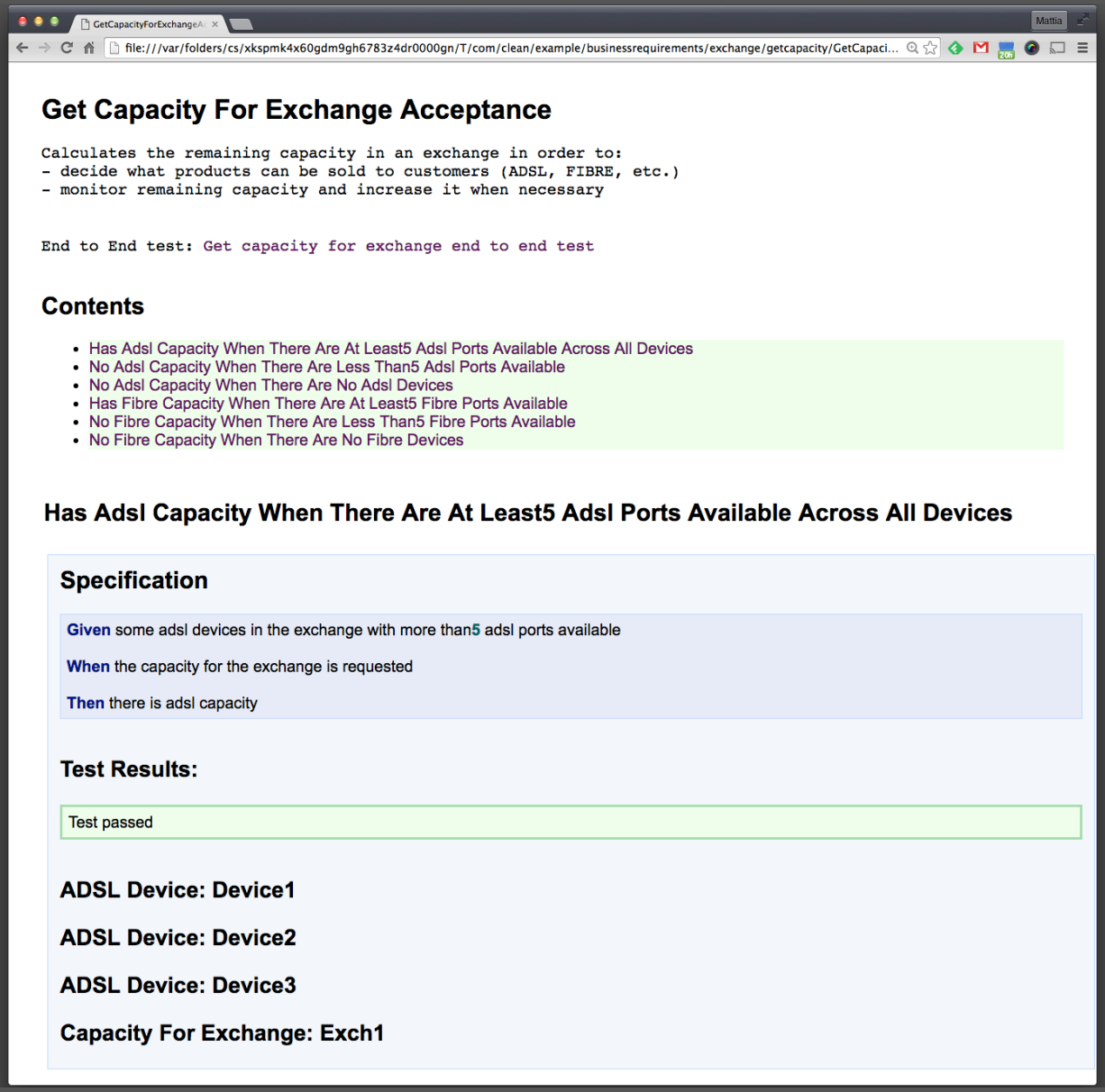
ACCEPTANCE TESTS



ACCEPTANCE TESTS

```
@Notes("Calculates the remaining capacity in an exchange...")
@LinkingNote(message = "End to End test: %s", links =
{GetCapacityForExchangeEndToEndTest.class})
public class GetCapacityForExchangeAcceptanceTest extends YatspecTest {

   // …mock the dependencies of the use case…

   GetCapacityForExchangeUseCase useCase = // …my use case

   @Test
   public void
hasAdslCapacityWhenThereAreAtLeast5AdslPortsAvailableAcrossAllDevices() {
      givenSomeAdslDevicesInTheExchangeWithMoreThan5AdslPortsAvailable();

      whenTheCapacityForTheExchangeIsRequested();

      thenThereIsAdslCapacity();
   }

   // …
}
```

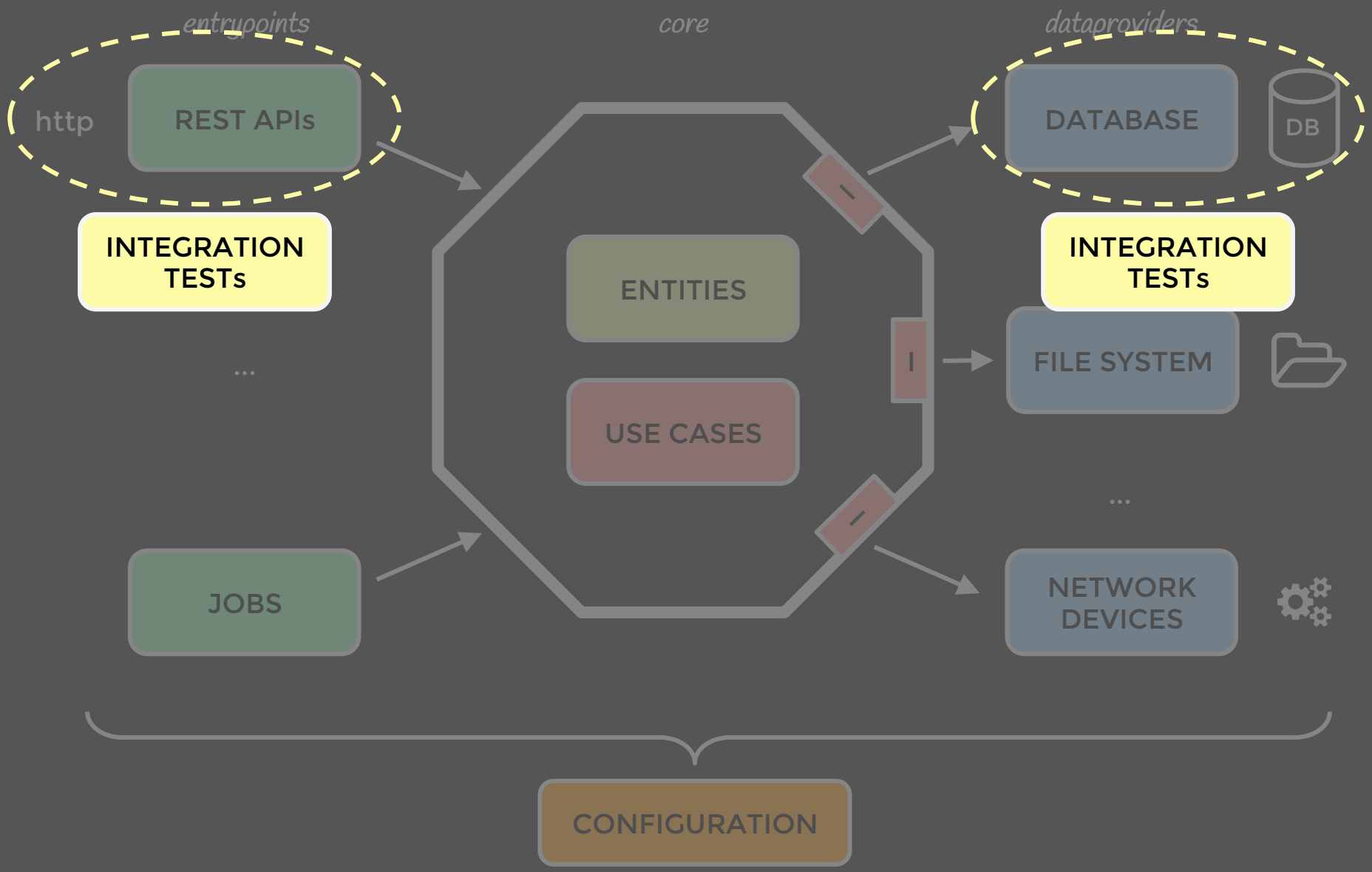
ACCEPTANCE TESTS



ACCEPTANCE TESTS

- BDD
  A.k.a. Conversations with the business
- Demonstrate business requirements
  A.k.a. Aim to cover business scenarios
- “Business” documentation
  A.k.a. What does the system do?
- Test use case in isolation, very fast
  A.k.a. No GUI, no DB, etc.
- Use your favourite BDD framework
  We use *Yatspec*

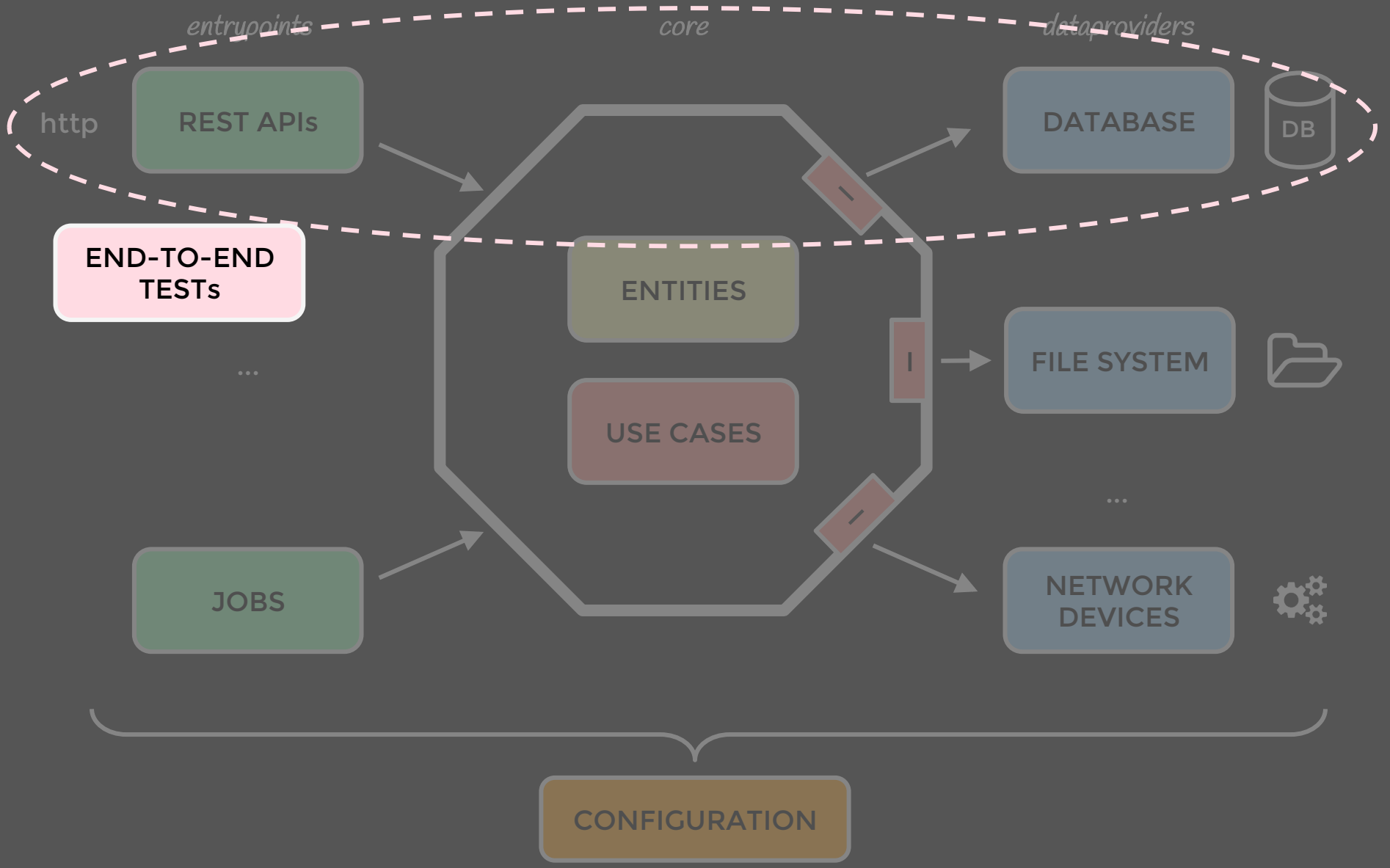
INTEGRATION TESTS



INTEGRATION TESTS

- Test integration with slow parts
  e.g. Http, database
- “Dev” documentation
  A.k.a. Does this work as expected?
- Test one layer in isolation
  e.g. only rest endpoint, or only data provider
- Use whatever library makes it easy
  e.g. Spring MockMVC; in-memory db

END-TO-END TESTS



END-TO-END TESTS

- Test only the critical journeys
  e.g. most common happy path
- Demonstrate “business” end-to-end requirement
- Start the whole app, very slow
  A.k.a. keep these to a minimum

PRO: TESTING STRATEGY

EFFECTIVE

TESTING STRATEGY

PROBLEMS? FIX!

Decisions taken too early

Hard to change

Infrequent deploys

Centered around frameworks

Centered around database

Business logic is spread everywhere

Hard to find things

Focused on technical aspects

Slow, heavy tests

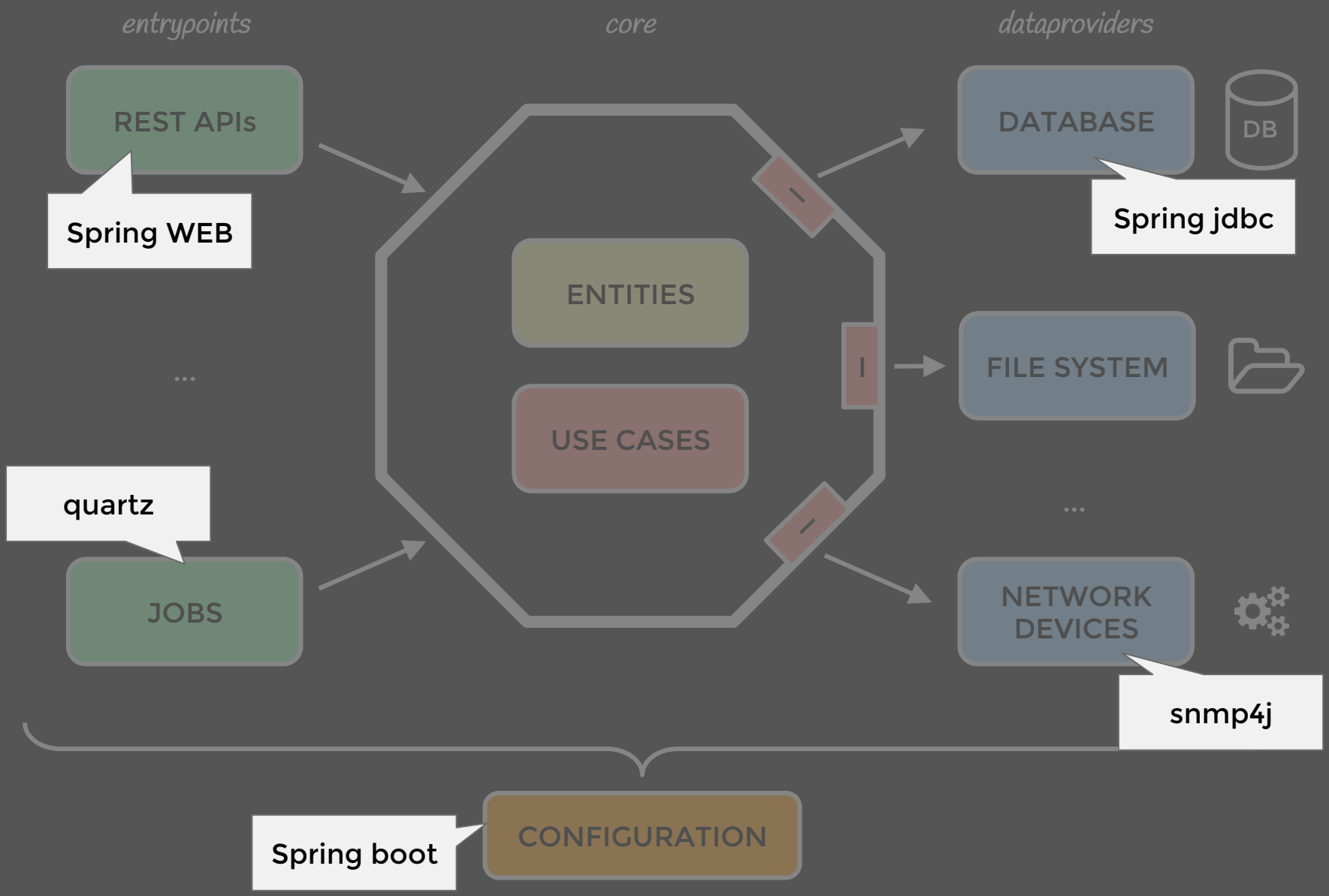
Effective Testing Pyramid

PRO: FRAMEWORKS

FRAMEWORKS ARE ISOLATED

easy(er) to change our mind

PRO: FRAMEWORKS



PROBLEMS? FIX!

Decisions taken too early

Hard to change

Infrequent deploys

Centered around frameworks

Centered around database

frameworks are Isolated

Business logic is spread everywhere

Hard to find things

Focused on technical aspects

Slow, heavy tests

Effective Testing Pyramid

PRO: DATABASE

INDEPENDENT
FROM DATABASE

PRO: DATABASE

- No CRUD!
- RESTful style for reading information
  GET /exchange/{exchangeCode}
  GET /exchange/{exchangeCode}/capacity
  GET /broadbandAccessDevice/{hostname}
- Custom endpoints for business actions
  POST /broadbandAccessDevice/{hostname}/makeLive
  PUT /exchange/{exchangeCode}/increaseCapacity

PROBLEMS? FIX!

Decisions taken too early

Hard to change

Infrequent deploys

Centered around frameworks

Centered around database

frameworks are Isolated

Independent from Database

Business logic is spread everywhere

Hard to find things

Focused on technical aspects

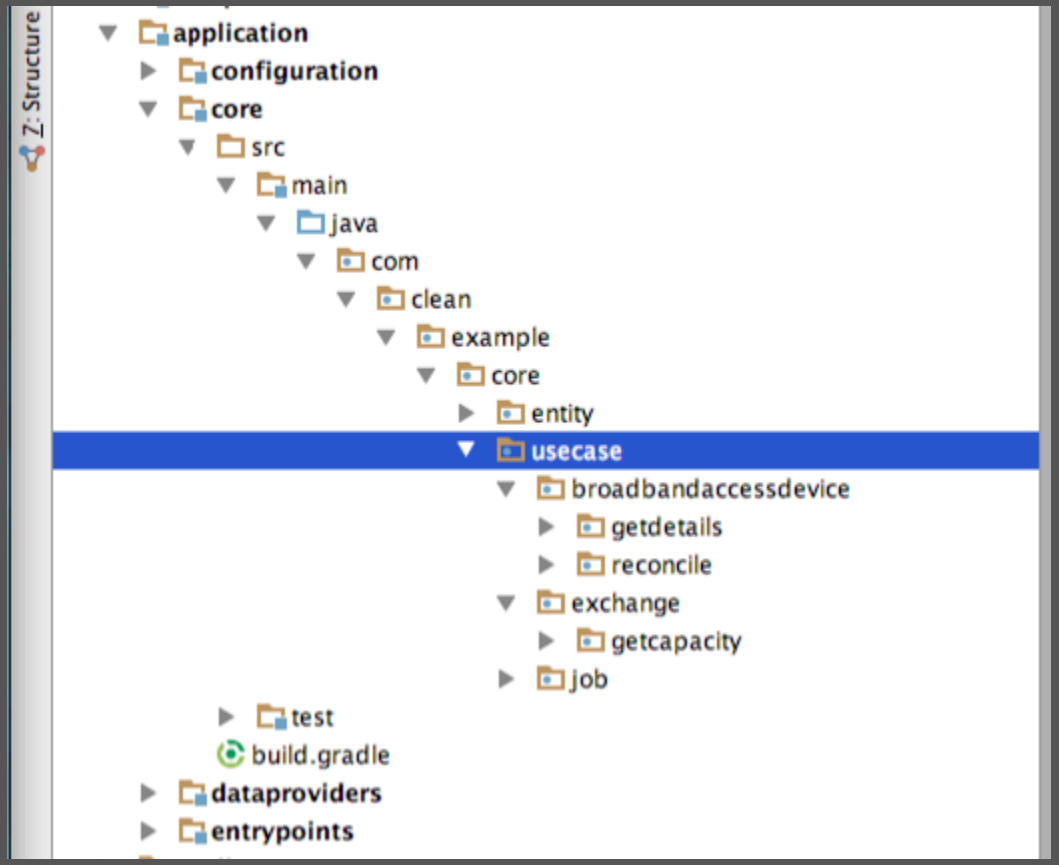
Slow, heavy tests

Effective Testing Pyramid

PRO: SCREAMING ARCHITECTURE

SCREAMS THE
INTENDED USAGE

PRO: SCREAMING ARCHITECTURE



PROBLEMS? FIX!

Decisions taken too early

Hard to change

Infrequent deploys

Centered around frameworks

Centered around database

frameworks are Isolated

Independent from Database

Business logic is spread everywhere

Hard to find things

Focused on technical aspects

Screaming Architecture

Slow, heavy tests

Effective Testing Pyramid

PRO: EASY TO FIND THINGS

ALL BUSINESS LOGIC IS IN

USE CASE

PROBLEMS? FIX!

Decisions taken too early

Hard to change

Infrequent deploys

Centered around frameworks

Centered around database

frameworks are Isolated

Independent from Database

Business logic is spread everywhere

All business logic in use cases

Hard to find things

Focused on technical aspects

Screaming Architecture

Slow, heavy tests

Effective Testing Pyramid

PRO: MODULES

HARD TO DO THE WRONG THING

PROBLEMS? FIX!

Decisions taken too early

Hard to change

Modules isolate changes

Infrequent deploys

Centered around frameworks

Centered around database

frameworks are Isolated

Independent from Database

Business logic is spread everywhere

All business logic in use cases

Hard to find things

Focused on technical aspects

Screaming Architecture

Slow, heavy tests

Effective Testing Pyramid

PRO: ALWAYS READY

ALWAYS READY TO DEPLOY

real continuous integration

PROBLEMS? FIX!

Decisions taken too early

Hard to change

Modules isolate changes

Infrequent deploys

Always Ready

Centered around frameworks

Centered around database

frameworks are Isolated

Independent from Database

Business logic is spread everywhere

All business logic in use cases

Hard to find things

Focused on technical aspects

Screaming Architecture

Slow, heavy tests

Effective Testing Pyramid

PRO: SWARMING

PARALLELISE WORK

multiple pairs on the same story

PROBLEMS? FIX!

Decisions taken too early

Hard to change

Modules isolate changes

Infrequent deploys

Always Ready

Centered around frameworks

Centered around database

frameworks are Isolated

Independent from Database

Business logic is spread everywhere

All business logic in use cases

Hard to find things

Focused on technical aspects

Screaming Architecture

Slow, heavy tests

Effective Testing Pyramid

Swarming

PRO: NICE MONOLITH

GOOD MONOLITH

TO START WITH

clear boundaries

to break on

PROBLEMS? FIX!

Decisions taken too early

Hard to change

Modules isolate changes

Infrequent deploys

Always Ready

Centered around frameworks

Centered around database

frameworks are Isolated

Independent from Database

Business logic is spread everywhere

All business logic in use cases

Hard to find things

Focused on technical aspects

Screaming Architecture

Slow, heavy tests

Effective Testing Pyramid

Nice Monolith

Swarming

CON: DUPLICATION

PERCEIVED DUPLICATION OF CODE

be pragmatic

PROBLEMS? FIX!

Decisions taken too early

Hard to change

Modules isolate changes

Infrequent deploys

Always Ready

Centered around frameworks

Centered around database

frameworks are Isolated

Independent from Database

Business logic is spread everywhere

All business logic in use cases

Hard to find things

Focused on technical aspects

Screaming Architecture

Duplication

Slow, heavy tests

Effective Testing Pyramid

Nice Monolith

Swarming

CON: IT NEEDS LOGIC

YOU NEED

INTERESTING

BUSINESS LOGIC

CON: IT NEEDS LOGIC

ENDPOINT

```
// …
@RequestMapping(value = API_PATH, method = GET)
public BroadbandAccessDeviceDto getDetails(@PathVariable String hostname) {
    try {
        BroadbandAccessDevice deviceDetails =
             useCase.getDeviceDetails(hostname);
        return toDto(deviceDetails);
    } catch (DeviceNotFoundException e) {
        throw new NotFoundException();
    }
}
// …
```

USE CASE

```
// …
public BroadbandAccessDevice getDeviceDetails(String hostname) {
    BroadbandAccessDevice device = dataProvider.getDetails(hostname);
    return device;
}
// …
```

DATA PROVIDER

```
// …
return "SELECT ...";
```

PROBLEMS? FIX!

Decisions taken too early

Hard to change

Modules isolate changes

Infrequent deploys

Always Ready

Centered around frameworks

Centered around database

frameworks are Isolated

Independent from Database

Business logic is spread everywhere

All business logic in use cases

Hard to find things

Focused on technical aspects

Screaming Architecture

Duplication

Slow, heavy tests

Effective Testing Pyramid

Nice Monolith

Swarming

Needs Interesting Logic

KEY TAKEAWAY

> The center of your application is the use cases of your application
>
> *Unclebob*

RESOURCES

Example Project (with all code shown in the presentation)

- Clean Architecture Example
  https://github.com/mattia-battiston/clean-architecture-example

Blogs & Articles

- The Clean Architecture
  https://blog.8thlight.com/uncle-bob/2012/08/13/the-clean-architecture.html
- Screaming Architecture
  http://blog.8thlight.com/uncle-bob/2011/09/30/Screaming-Architecture.html
- NODB
  https://blog.8thlight.com/uncle-bob/2012/05/15/NODB.html
- Hexagonal Architecture
  http://alistair.cockburn.us/Hexagonal+architecture

Videos & Presentations

- Clean Coders ep. 7: Architecture, Use Cases, and High Level Design
  https://cleancoders.com/episode/clean-code-episode-7/show
- Robert C. Martin - Clean Architecture
  https://vimeo.com/43612849
- Robert C. Martin - Clean Architecture and Design
  https://www.youtube.com/watch?v=Nsjsiz2A9mg

THANK YOU!

@BattistonMattia

mattia.battiston@gmail.com

mattia-battiston

really, really appreciated! Help me improve