By Petter Holmström

PART OF TUTORIAL SERIES

Domain Driven Design Crash Course

Domain-Driven Design and the Hexagonal Architecture

In the two previous articles, we learned about strategic and tactical domain-driven design. Now it is time to learn how to turn a domain model into working software - more specifically, how to do it using the hexagonal architecture.

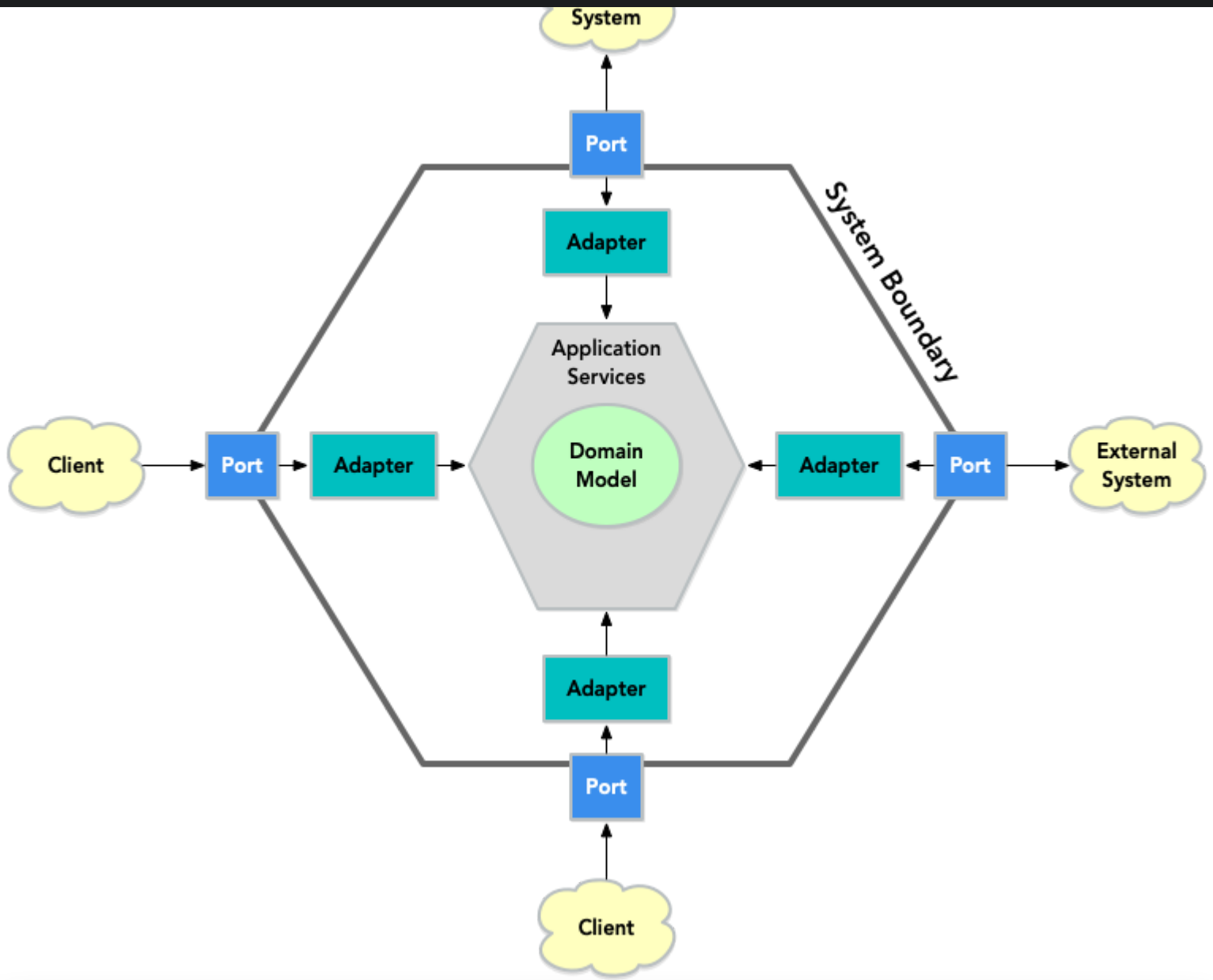
The previous two articles were pretty generic even though the code examples were written in Java. Although a lot of the theory in this article can also be applied in other environment and languages as well, I have explicitly written it with Java and Vaadin in mind.

Again, the content is based on the books **Domain-Driven Design: Tackling Complexity in the Heart of Software** by Eric Evans and **Implementing Domain-Driven Design** by **Vaughn Vernon** and I highly

believe. That said, it was the books of Evans and Vernon that got me started with DDD in the first place and I'd like to think that what I'm writing here is not too far from what you will find in the books.

Why Is It Called Hexagonal?

The name *hexagonal architecture* comes from the way this architecture is usually depicted:

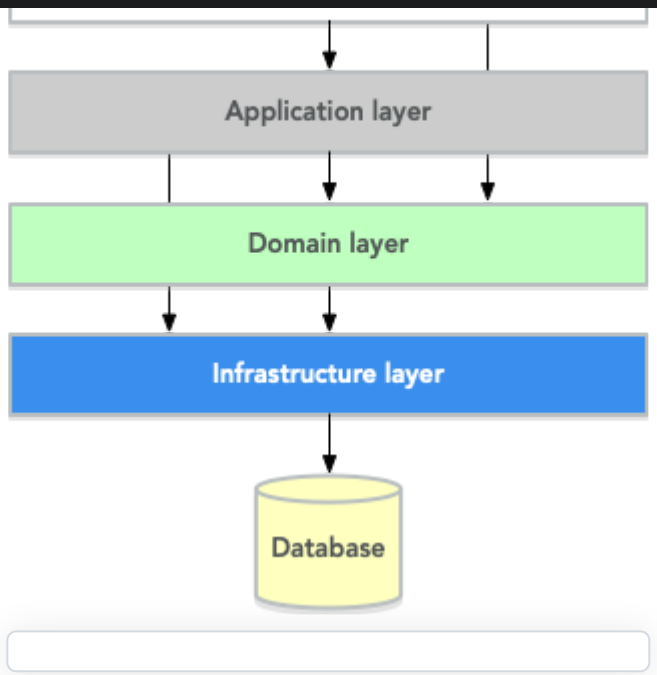

System
Port
Adapter
System Boundary
Application Services
Client
Port
Adapter
Domain Model
Adapter
Port
External System
Adapter
Port
Client

onion architecture because of how it is layered.

In the following, we are going to have a closer look at the "onion". We are going to start with the core - the domain model - and then work ourselves out, one layer at the time, until we reach the ports and the systems and clients interacting with them.

Hexagonal vs. Traditional Layers

Once we dig deeper into the hexagonal architecture you will find that it has several resemblances to the more traditional layered architecture. Indeed, you can think of the hexagonal architecture as an evolution of the layered architecture. However, there are some differences especially with regards to how a system interacts with the outside world. To better understand these differences, let’s start with a recap of the layered architecture:



The principle is that the system is built up of layers stacked on top of each other. A higher layer can interact with a lower layer *but not the other way around*. Typically, in a domain-driven layered architecture, you would have the UI layer at the top. This layer, in turn, interacts with an application service layer, which interacts with the domain model that lives in a domain layer. At the bottom, we have an infrastructure layer that communicates with external systems such as a database.

In the hexagonal system, you will find that the application layer and the domain layer are still pretty much the same. However, the UI layer and the infrastructure layer are treated in a very different way. Keep reading to find out how.

The Domain Model

lives, where all the business decisions are made. This is also the most stable part of the software that hopefully will change the least (unless the business itself changes of course).

The domain model has been the subject of the previous two articles in this series, so we are not going to cover it anymore here. However, the domain model alone does not provide any value if there is no way of interacting with it. To do that, we have to move up to the next layer in the "onion".

Application Services

An application service acts as a facade through which clients will interact with the domain model. Application services have the following characteristics:

- They are stateless
- They enforce system security
- They control the database transactions
- They orchestrate business operations but do not make any business decisions (i.e., they do not contain any business logic)

Let’s have a closer look at what this means.

Hexagonal vs. Entity-Control-Boundary

If you have heard of the **Entity-Control-Boundary** pattern before, you will find the hexagonal architecture familiar. You can think of your aggregates as **entities**, domain services, factories and repositories as **controllers** and the application services as **boundaries**.

An application service does not maintain any internal state that can be changed by interacting with clients. All the information that is needed to perform an operation should be available as input parameters to the application service method. This will make the system simpler and easier to debug and scale.

If you find yourself in a situation where you have to make multiple application service calls within the context of a single business process, you can model the business process in a class of its own and pass an instance of it as an input parameter to the application service method. The method would then do its magic and return an updated instance of the business process object that in turn can be used as input to other application service methods:

Business Process as Input Argument

```
public class MyBusinessProcess {
    // Current process state
}

public interface MyApplicationService {

    MyBusinessProcess performSomeStuff(MyBusinessProcess input);

    MyBusinessProcess performSomeMoreStuff(MyBusinessProcess input);
}
```

You could also make the business process object mutable and let the application service method change the state of the object directly. I personally do not prefer this approach since I believe it can lead to unwanted side effects, especially if the transaction ends up rolling back. This depends on how

For tips on how to implement more complex and long-running business processes, I encourage you to read Vernon's book.

Security Enforcement

The application service makes sure that the current user is allowed to perform the operation in question. Technically, you can do this manually at the top of each application service method or use something more sophisticated such as AOP. It does not matter how security is enforced as long as it happens in the application service layer and not inside the domain model. Now, why is this important?

When we talk about security in an application, we tend to put more emphasis on preventing unauthorized access than on permitting authorized access. Thus, any security check we add to the system will essentially make it harder to use. If we add these security checks to the domain model, we may find ourselves in a situation where we are unable to perform an important operation because we did not think of it when the security checks were added and now they stand in the way. By keeping the all security checks out of the domain model, we get a more flexible system since we can interact with the domain model in any way we want. The system will still be safe since all clients are required to go through an application service anyway. It is way easier to create a new application service than to change the domain model.

CODE EXAMPLES

Here are two Java-examples of what security enforcement in an application service could look like. The code has not been tested and should be treated more as pseudo-code than actual Java code.

Declarative Security Enforcement

```
    @Secured("ROLE_BUSINESS_PROCESSOR") // (1)
    public MyBusinessProcess performSomeStuff(MyBusinessProcess input) {
        var customer = customerRepository.findById(input.getCustomerId()) // (2)
            .orElseThrow( () -> new CustomerNotFoundException(input.getCustomerId()));
        var someResult = myDomainService.performABusinessOperation(customer); // (3)
        customer = customerRepository.save(customer);
        return input.updateMyBusinessProcessWithResult(someResult); // (4)
    }
}
```

1. The annotation instructs the framework to only allow authenticated users with the role `ROLE_BUSINESS_PROCESSOR` to invoke the method.
2. The application service looks up an aggregate root from a repository in the domain model.
3. The application service passes the aggregate root to a domain service in the domain model, storing the result (whatever it is).
4. The application service uses the result of the domain service to update the business process object and returns it so that it can be passed to other application service methods participating in the same longrunning process.

Manual Security Enforcement

```
    public MyBusinessProcess performSomeStuff(MyBusinessProcess input) {
        // We assume SecurityContext is a thread-local class that contains information
        // about the current user.
        if (!SecurityContext.isLoggedOn()) { // (1)
            throw new AuthenticationException("No user logged on");
        }
        if (!SecurityContext.holdsRole("ROLE_BUSINESS_PROCESSOR")) { // (2)
            throw new AccessDeniedException("Insufficient privileges");
        }

        var customer = customerRepository.findById(input.getCustomerId())
            .orElseThrow( () -> new CustomerNotFoundException(input.getCustomerId()));
        var someResult = myDomainService.performABusinessOperation(customer);
        customer = customerRepository.save(customer);
        return input.updateMyBusinessProcessWithResult(someResult);
    }
}
```

1. In a real application, you would probably create helper methods that throw the exception if a user is not logged on. I have only included a more verbose version in this example to show what needs to be checked.
2. As in the previous case, only users with the role `ROLE_BUSINESS_PROCESSOR` are allowed to invoke the method.

Transaction Management

service method succeeds, there is no way of undoing it except by explicitly invoking another application service that reverses the operation (if such a method even exists).

If you find yourself in a situation where you would want to invoke multiple application service methods within the same transaction, you should check that the granularity of your application service is correct. Maybe some of the things your application service is doing should actually be in domain services instead? You may also need to consider redesigning your system to use eventual consistency instead of strong consistency (for more information about this, please check the previous article about tactical domain-driven design).

Technically, you can either handle the transactions manually inside the application service method or you can use the declarative transactions that are offered by frameworks and platforms such as Spring and Java EE.

CODE EXAMPLES

Here are two Java-examples of what transaction management in an application service could look like. The code has not been tested and should be treated more as pseudo-code than actual Java code.

Declarative Transaction Management

```
    @Transactional // (1)
    public void resetPassword(UserId userId) {
        var user = userRepository.findByUserId(userId); // (2)
        user.resetPassword(); // (3)
        userRepository.save(user);
    }
}
```

1. The framework will make sure the entire method runs inside a single transaction. If an exception is thrown, the transaction is rolled back. Otherwise, it is committed when the method returns.
2. The application service calls a repository in the domain model to find the `User` aggregate root.
3. The application service invokes a business method on the `User` aggregate root.

Manual Transaction Management

```
    @Transactional
    public void resetPassword(UserId userId) {
        var tx = transactionManager.begin(); // (1)
        try {
            var user = userRepository.findByUserId(userId);
            user.resetPassword();
            userRepository.save(user);
            tx.commit(); // (2)
        } catch (RuntimeException ex) {
            tx.rollback(); // (3)
            throw ex;
        }
    }
}
```

1. The transaction manager has been injected into the application service so that the service method can start a new transaction explicitly.
2. If everything works, the transaction is committed after the password has been reset.
3. If an error occurs, the transaction is rolled back and the exception is rethrown.

Orchestration

Getting the orchestration right is perhaps the most difficult part of designing a good application service. This is because you need to make sure you are not accidentally introducing business logic into

By orchestration, I mean looking up and invoking the correct domain objects in the correct order, passing in the correct input parameters and returning the correct output. In its simplest form, an application service may look up an aggregate based on an ID, invoke a method on that aggregate, save it and return. However, in more complex cases, the method may have to look up multiple aggregates, interact with domain services, perform input validation and so on. If you find yourself writing long application service methods, you should ask yourself the following questions:

- Is the method making a business decision or asking the domain model to make the decision?
- Should some of the code be moved to domain event listeners?

This being said, having some business logic ending up in an application service method is not the end of the world. It is still pretty close to the domain model and well encapsulated and should be pretty easy to refactor into the domain model at a later time. Don't waste too much precious time thinking about whether something should go into the domain model or into the application service if it is not immediately clear to you.

CODE EXAMPLES

Here is a Java-example of what a typical orchestration could look like. The code has not been tested and should be treated more as pseudo-code than actual Java code.

Orchestration Involving Multiple Domain Objects

```
    @Transactional // (1)
    @PermitAll // (2)
    public Customer registerNewCustomer(CustomerRegistrationRequest request) {
        var violations = validator.validate(request); // (3)
        if (violations.size() > 0) {
            throw new InvalidCustomerRegistrationRequest(violations);
        }
        customerDuplicateLocator.checkForDuplicates(request); // (4)
        var customer = customerFactory.createNewCustomer(request); // (5)
        return customerRepository.save(customer); // (6)
    }
}
```

1. The application service method runs inside a transaction.
2. The application service method can be accessed by any user.
3. We invoke a JSR-303 validator to check that the incoming registration request contains all the necessary information. If the request is invalid, we throw an exception that will be reported back to the user.
4. We invoke a domain service that will check if there already is a customer in the database with the same information. If that is the case, the domain service will throw an exception (not shown here) that will be propagated back to the user.
5. We invoke a domain factory that will create a new `Customer` aggregate with information from the registration request object.

Domain Event Listeners

In the previous article about tactical domain-driven design, we talked about domain events and domain event listeners. We did not, however, talk about where the domain event listeners fit into the overall system architecture. We recall from the previous article that a domain event listener should not be able to affect the outcome of the method that published the event in the first place. In practice, this means that a domain event listener should run inside its own transaction.

Because of this, I consider domain event listeners to be a special kind of application service that is invoked not by a client but by a domain event. This also means that a domain event listener is an orchestrator that should not contain any business logic. Depending on what needs to happen when a certain domain event is published, you may have to create a separate domain service that decides what to do with it if there is more than one path forward.

This being said, in the section about aggregates in the previous article, I mentioned that it may sometimes be justified to alter multiple aggregates within the same transaction even though this goes against the aggregate design guidelines. I also mentioned that this should preferably be made through domain events. In cases like this, the domain event listeners would have to participate in the current transaction and could thereby affect the outcome of the method that published the event, breaking the design guidelines for both domain events and application services. This is not the end of the world as long as you do it intentionally and are aware of the consequences you might face in the future. Sometimes you just have to be pragmatic.

Input and Output

One important decision when designing application services is to decide what data to consume (method parameters) and what data to return. You have three alternatives:

3. Use Domain Payload Objects (DPOs) that are a combination of the two above.

Each alternative has its own pros and cons, so let's have a closer look at each.

ENTITIES AND AGGREGATES

In the first alternative, the application services return entire aggregates (or parts thereof). The client can do whatever it wants with them and when it is time to save changes, the aggregates (or parts thereof) are passed back to the application service as parameters.

This alternative works best when the domain model is anemic (i.e. it only contains data and no business logic) and the aggregates are small and stable (as in unlikely to change much in the near future).

It also works if the client will be accessing the system through REST or SOAP and the aggregates can easily be serialized into JSON or XML and back. In this case, clients will not actually be interacting directly with your aggregates but with a JSON or XML representation of the aggregate that may be implemented in a completely different language. From the client's perspective, the aggregates are just DTOs.

The advantages of this alternative are:

- You can use the classes that you already have
- There is no need to convert between domain objects and DTOs.

The disadvantages are:

- It imposes restrictions on how you validate user input (more about this later).
- You have to design your aggregates in such a way that the client cannot put the aggregate into an inconsistent state or perform an operation that is not allowed.
- You may run into problems with lazy-loading of entities inside an aggregate (JPA).

DATA TRANSFER OBJECTS

In the second alternative, the application services consume and return data transfer objects. The DTOs can correspond to entities in the domain model, but more often they are designed for a specific application service or even a specific application service method. The application service is then responsible for moving data back and forth between the DTOs and the domain objects.

This alternative works best when the domain model is very rich in business logic, the aggregates are complex or when the domain model is expected to change a lot while keeping the client API as stable as possible.

The advantages of this alternative are:

- The clients are decoupled from the domain model, making it easier to evolve it without having to change the clients.
- Only the data that is actually needed is being passed between the clients and the application services, improving performance (especially if the client and the application service are communicating over a network in a distributed environment).
- It becomes easier to control access to the domain model, especially if only certain users are allowed to invoke certain aggregate methods or view certain aggregate attribute values.

The disadvantages are:

- You get a new set of DTO classes to maintain.
- You have to move data back and forth between DTOs and aggregates. This can be especially tedious if the DTOs and entities are almost similar in structure. If you work in a team you need to have a good explanation ready for why the separation of DTOs and aggregates is warranted.

DOMAIN PAYLOAD OBJECTS

In the third alternative, application services consume and return domain payload objects. A domain payload object is a data transfer object that is aware of the domain model and can contain domain objects. This is essentially a combination of the first two alternatives.

This alternative works best in cases where the domain model is anemic, the aggregates are small and stable and you want to implement an operation that involves multiple different aggregates. Personally, I would say I use DPOs more often as output objects than as input objects.

The advantages of this alternative are:

- You do not need to create DTO classes for everything. When passing a domain object directly to the client is good enough, you do it. When you need a custom DTO, you create one. When you need both, you use both.

The disadvantages are:

CODE EXAMPLES

Here are two Java examples of using DTOs and DPOs, respectively. The DTO example demonstrates a use case where it makes sense to use a DTO than return the entity directly: Only a fraction of the entity attributes are needed and we need to include information that does not exist in the entity. The DPO example demonstrates a use case where it makes sense to use a DPO: We need to include many different aggregates that are related to each other in some way.

The code has not been tested and should be treated more as pseudo-code than actual Java code.

Data Transfer Object Example

```
    private String name;
    private LocalDate lastInvoiceDate;

    // Getters and setters omitted
}

@Service
public class CustomerListingService {

    @Transactional
    public List<CustomerListEntryDTO> getCustomerList() {
        var customers = customerRepository.findAll(); // (2)
        var dtos = new ArrayList<CustomerListEntryDTO>();
        for (var customer : customers) {
            var lastInvoiceDate = invoiceService.findLastInvoiceDate(customer.getId()); //
            dto = new CustomerListEntryDTO(); // (4)
            dto.setId(customer.getId());
            dto.setName(customer.getName());
            dto.setLastInvoiceDate(lastInvoiceDate);
            dtos.add(dto);
        }
        return dto;
    }
}
```

last invoice date.

2. We look up all the customer aggregates from the database. In a real-world application, this would be a paginated query that only returns a subset of the customers.
3. The last invoice date is not stored in the customer entity so we have to invoke a domain service to look it up for us.
4. We create the DTO instance and populate it with data.

Domain Payload Object Example

```
    private YearMonth month;
    private Collection<Invoice> invoices;

    // Getters and setters omitted
}

@Service
public class CustomerInvoiceSummaryService {

    public CustomerInvoiceMontlySummaryDPO getMonthlySummary(CustomerId customerId, YearMo
        var customer = customerRepository.findById(customerId); // (2)
        var invoices = invoiceRepository.findByYearMonth(customerId, month); // (3)
        var dpo = new CustomerInvoiceMonthlySummaryDPO(); // (4)
        dpo.setCustomer(customer);
        dpo.setMonth(month);
        dpo.setInvoices(invoices);
        return dpo;
    }
}
```

1. The Domain Payload Object is a data structure without any business logic that contains both domain objects (in this case entities) and additional information (in this case the year and month).
2. We fetch the customer's aggregate root from the repository.
3. We fetch the customer's invoices for the specified year and month.
4. We create the DPO instance and populate it with data.

As we have mentioned previously, an aggregate must always be in a consistent state. This means among other things that we need to properly validate all the input that is used to alter the state of an aggregate. How and where do we do that?

From a user experience perspective, the user interface should include validation so that the user is not even able to perform an operation if the data is invalid. However, relying simply on user interface validation is *not good enough* in a hexagonal system. The reason for this is that the user interface is but one of potentially many ports into the system. It does not help that the user interface is validating data properly if a REST endpoint lets any garbage through to the domain model.

When thinking about input validation there are actually two distinct kinds of validation: format validation and content validation. When we are validating the format, we check that certain values of certain types conform to certain rules. E.g. a social security number is expected to be in a specific pattern. When we are validating the content, we already have a well-formed piece of data and are interested in checking that that data makes sense. E.g. we may want to check that a well-formed social security number actually corresponds to a real person. You can implement these validations in different ways so let's have a closer look.

FORMAT VALIDATION

If you are using a lot of value objects in your domain model (I tend to do that personally) that are wrappers around primitive types (such as strings or integers), then it makes sense to build the format validation straight into your value object constructor. In other words, it should not be possible to create e.g. an `EmailAddress` or `SocialSecurityNumber` instance without passing in a well-formed argument. This has the added advantage that you can do some parsing and cleaning up inside the constructor if there are multiple known ways of entering valid data (e.g. when entering a phone number some

Now when the value objects are valid, how do we validate the entities that use them? There are two options available for Java developers.

The first option is to add the validation into your constructors, factories and setter methods. The idea here is that it should not even be possible to put an aggregate into an inconsistent state: all required fields must be populated in the constructor, any setters of required fields will not accept null parameters, other setters will not accept values of an incorrect format or length, etc. I personally tend to use this approach when I'm working with domain models that are very rich in business logic. It makes the domain model very robust, but also practically forces you to use DTOs between clients and application services since it is more or less impossible to properly bind to a UI.

The second option is to use Java Bean Validation (JSR-303). Put annotations on all of the fields and make sure your application service runs the aggregate through the `Validator` before doing anything else with it. I personally tend to use this approach when I'm working with domain models that are anemic. Even though the aggregate itself does not prevent anybody from putting it into an inconsistent state, you can safely assume that all aggregates that have either been retrieved from a repository or have passed validation are consistent.

You can also combine both options by using the first option in your domain model and Java Bean Validation for your incoming DTOs or DPOs.

CONTENT VALIDATION

The simplest case of content validation is to make sure that two or more interdependent attributes within the same aggregate are valid (e.g. if one attribute is set, the other must be null and vice versa). You can either implement this directly into the entity class itself or use a class-level Java Bean

A more complex case of content validation would be to check that a certain value exists (or does not exist) in a lookup list somewhere. This is very much the responsibility of the application service. Before allowing any business or persistence operations to continue, the application service should perform the lookup and throw an exception if needed. This is not something you want to put into your entities since the entities are movable domain objects whereas the objects needed for the lookup are typically static (see the previous article about tactical DDD for more information about movable and static objects).

The most complex case of content validation would be to verify an entire aggregate against a set of business rules. In this case, the responsibility is split between the domain model and the application service. A domain service would be responsible for performing the validation itself, but the application service would be responsible for invoking the domain service.

CODE EXAMPLES

Here we are going to look at three different ways of handling validation. In the first case, we will look at performing format validation inside the constructor of a value object (a phone number). In the second case, we will look at an entity that has validation built-in so that it is not possible to put the object into an inconsistent state in the first place. In the third and last case, we will look at the same entity but implemented using JSR-303 validation. That makes it possible to put the object into an inconsistent state, but not to save it to the database as such.

Value Object with Format Validation

```
    public PhoneNumber(String phoneNumber) {
        Objects.requireNonNull(phoneNumber, "phoneNumber must not be null"); // (1)
        var sb = new StringBuilder();
        char ch;
        for (int i = 0; i < phoneNumber.length(); ++i) {
            ch = phoneNumber.charAt(i);
            if (Character.isDigit(ch)) { // (2)
                sb.append(ch);
            } else if (!Character.isWhitespace(ch) && ch != '(' && ch != ')' && ch != '-'
                throw new IllegalArgument(phoneNumber + " is not valid");
            }
        }
        if (sb.length() == 0) { // (4)
            throw new IllegalArgumentException("phoneNumber must not be empty");
        }
        this.phoneNumber = sb.toString();
    }

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

    // Equals and hashCode omitted
}
```

numbers, we should support a '+' sign as the first character as well, but we'll leave that as an exercise to the reader.

3. We allow, but ignore, whitespace and certain special characters that people often use in phone numbers.
4. Finally, when all the cleaning is done, we check that the phone number is not empty.

Entity with Built-in Validation

```
    // Fields omitted

    public Customer(CustomerNo customerNo, String name, PostalAddress address) {
        setCustomerNo(customerNo); // (1)
        setName(name);
        setPostalAddress(address);
    }

    public setCustomerNo(CustomerNo customerNo) {
        this.customerNo = Objects.requireNonNull(customerNo, "customerNo must not be null"
    }

    public setName(String name) {
        Objects.requireNonNull(nanme, "name must not be null");
        if (name.length() < 1 || name.length > 50) { // (2)
            throw new IllegalArgumentException("Name must be between 1 and 50 characters"
        }
        this.name = name;
    }

    public setAddress(PostalAddress address) {
        this.address = Objects.requireNonNull(address, "address must not be null");
    }
}
```

subclass decides to override any of them. In this case, it would be better to mark the setter methods as final but some persistence frameworks may have a problem with that. You just have to know what you are doing.

2. Here we check the length of a string. The lower limit is a business requirement since every customer must have a name. The upper level is a database requirement since the database, in this case, has a schema that only allows it to store strings of 50 characters. By adding the validation here already, you can avoid annoying SQL errors at a later stage when you try to insert too long strings into the database.

Entity with JSR-303 Validation

```
public class Customer implements Entity {

    @NotNull (1)
    private CustomerNo customerNo;

    @NotBlank (2)
    @Size(max = 50) (3)
    private String name;

    @NotNull
    private PostalAddress address;

    // Setters omitted
}
```

1. This annotation ensures that the customer number cannot be null when the entity is saved.

saved.

Does the Size Matter?

Before we leave the subject of application services, there is one more thing I want to briefly mention. As with all facades, there is an ever-present risk of the application services growing into huge god classes that know too much and do too much. These types of classes are often hard to read and maintain simply because they are so large.

So how do you keep the application services small? The first step is of course to split a service that is growing too big into smaller services. However, there is a risk in this as well. I have seen situations were two services where so similar that developers did not know what the difference was between them, nor which method should go into which service. The result was that service methods were scattered over two separate service classes, and sometimes even implemented twice - once in each service - but by different developers.

When I design application services, I try to make them as coherent as possible. In CRUD applications, this could mean one application service per aggregate. In more domain-driven applications, this could mean one application service per business process or even separate services for specific use cases or user interface views.

Naming is a very good guideline when designing application services. Try to name your application services according to what they do as opposed to which aggregates they concern. E.g. `EmployeeCrudService` or `EmploymentContractTerminationService` are far better names than `EmployeeService` which could mean anything.

Finally, I just want to mention command based application services. In this case, you model each application service model as a command object with a corresponding command handler. This means

in a large number of small classes and is useful especially in applications whose user interfaces are inherently command-driven or where clients interact with application services through some kind of messaging mechanism such as a message queue (MQ) or enterprise service bus (ESB).

CODE EXAMPLES

I'm not going to give you an example of what a God-class looks like because that would take up too much space. Besides, I think most developers who have been in the profession for a while have seen their fair share of such classes. Instead, we are going to look at an example of what a command based application service could look like. The code has not been tested and should be treated more as pseudo-code than actual Java code.

Command Based Application Services

```
public interface Command<R> { // (1)
}

public interface CommandHandler<C extends Command<R>, R> { // (2)

    R handleCommand(C command);
}

public class CommandGateway { // (3)

    // Fields omitted

    public <C extends Command<R>, R> R handleCommand(C command) {
        var handler = commandHandlers.findHandlerFor(command)
                .orElseThrow(() -> new IllegalStateException("No command handler found"));
```

```
}

public class CreateCustomerCommand implements Command<Customer> { // (4)
    private final String name;
    private final PostalAddress address;
    private final PhoneNumber phone;
    private final EmailAddress email;

    // Constructor and getters omitted
}

public class CreateCustomerCommandHandler implements CommandHandler<CreateCustomerCommand

    @Override
    @Transactional
    public Customer handleCommand(CreateCustomerCommand command) {
        var customer = new Customer();
        customer.setName(command.getName());
        customer.setAddress(command.getAddress());
        customer.setPhone(command.getPhone());
        customer.setEmail(command.getEmail());
        return customerRepository.save(customer);
    }
}
```

2. The `CommandHandler` interface is implemented by a class that knows how to handle (perform) a particular command and return the result.
3. Clients interact with a `CommandGateway` to avoid having to lookup individual command handlers. The gateway knows about all available command handlers and how to find the correct one based on any given command. The code for looking up handlers is not included in the example since it depends on the underlying mechanism for registering handlers.
4. Every command implements the `Command` interface and includes all the necessary information to perform the command. I like to make my commands immutable with built-in validation, but you can also make them mutable and use JSR-303 validation.
5. Every command has its own handler that performs the command and returns the result.

Ports and Adapters

So far we have discussed the domain model and the application services that surround and interact with it. However, these application services are completely useless if there is no way for clients to invoke them and that is where ports and adapters enter the picture.

A port is an interface between the system and the outside world that has been specially designed for a particular type of protocol. Ports are not only used to allow outside clients access to the system but also to allow the system to access external systems.

Because ports can be designed for many different protocols, you have to pair them with adapters that know how to translate between the protocol and the application services and domain model.

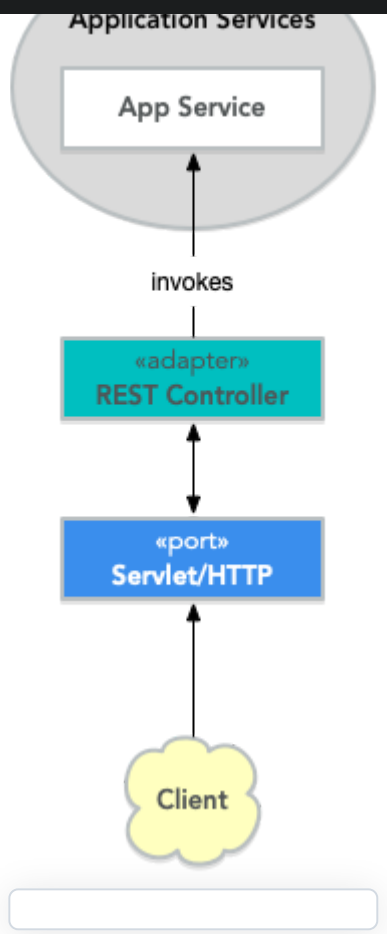
You typically have a one-to-one mapping between ports and adapters but nothing prevents you from using the same adapter with different ports. You could, for instance, make your REST API available

You can add as many ports and adapters as you like to a system and also remove obsolete ports and adapters that are no longer in use. This is one of the core strengths of the hexagonal architecture - it is extendable and flexible.

This may all sound like a good idea but what are ports and adapters in practice? Let's have a look!

Example 1: A REST API

In the first example we are going to create a REST API for our Java application:



The port is HTTP so we need a servlet to implement the port. The REST controller acts as the adapter. Naturally we are using a framework such as Spring or JAX-RS that provides both the servlet and mapping between POJOs (Plain Old Java Objects) and XML/JSON out-of-the-box. We only have to implement the REST controller which will:

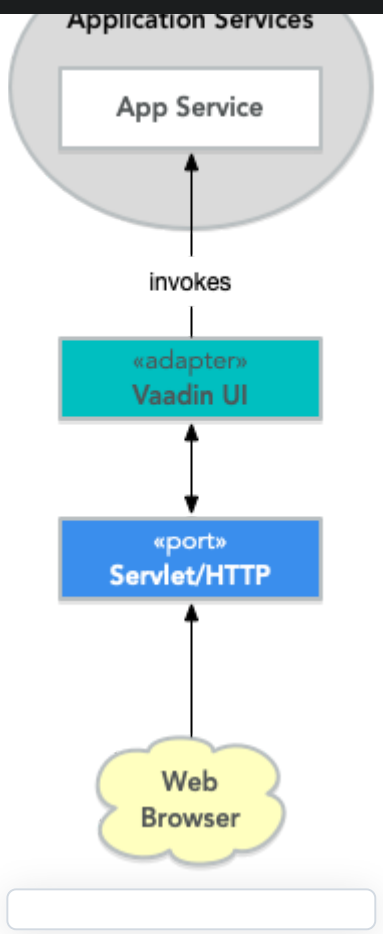
1. Take either raw XML/JSON or deserialized POJOs as input,
2. Invoke the application services,

4. Return the response to the client.

The clients, regardless of whether they are client-side web applications running in a browser or other systems running on their own servers, are not a part of this particular hexagonal system. The system also does not have to care about who the clients are as long as they conform to the protocol that the port and adapter supports.

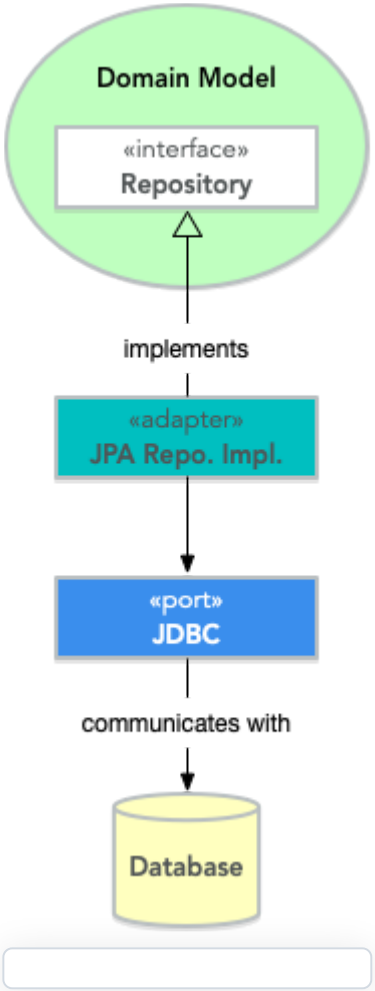
Example 2: A Server-Side Vaadin UI

In the second example, we are going to look at a different type of adapter, namely a server-side Vaadin UI:



Again, the port is HTTP so we need a servlet to implement the port. In this case, the servlet is the `VaadinServlet`, so we don't need to implement it ourselves (this is the case regardless of whether you are using Vaadin 8 or Flow). Now we just need an adapter for translating incoming user actions into application service method calls and the output into HTML that can be rendered in the browser. This adapter is the Vaadin UI. Thinking of the user interface as just another port/adapter into the system is an excellent way of keeping business logic outside of the user interface.

In the third example, we are going to turn things around and look at a port and adapter that allows our system to call out to an external system, more specifically a relational database:



This time, the port is JDBC and it is implemented by a JDBC driver (such as for H2, MySQL or PostgreSQL). The adapter is a set of implementations of the repository interfaces declared in the domain model. The adapter would also have to plug into the application service layer's transaction

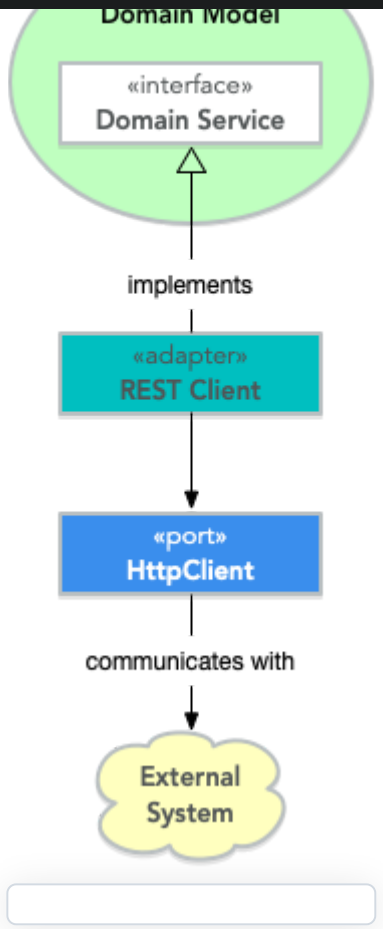
adapter.

In this case, the adapter is not invoking the application service. Instead, the application service is invoking the adapter, but through interfaces defined by the application service or domain model. For this to work properly, you need to use some kind of dependency injection so that the correct instances of the adapter classes are injected into the application services.

This is a use case that is perfectly OK in a hexagonal architecture but would not have been permitted in a traditional layered architecture. The reason for this is that we would have a lower layer (the "infrastructure layer") depending on a higher layer (the "domain layer" and the "application layer").

Example 4: Communicating with an External System over REST

In the fourth and last example, we are going to look at a port and adapter that allows our system to call out to an external system over REST:

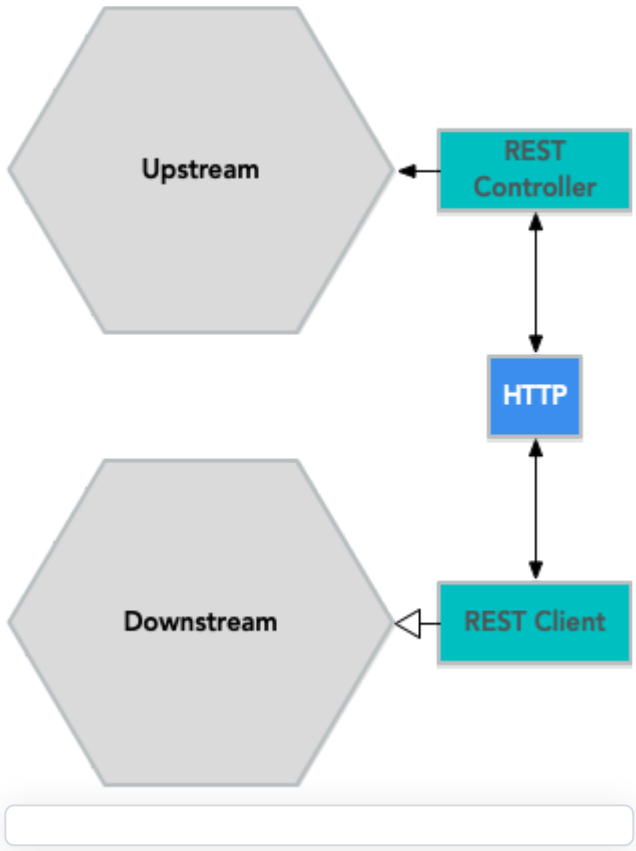


The port is again HTTP, but it is sending out requests and receiving responses instead of the other way around (as was the case in the first example). It is implemented by some suitable HTTP client, such as Apache HttpComponents.

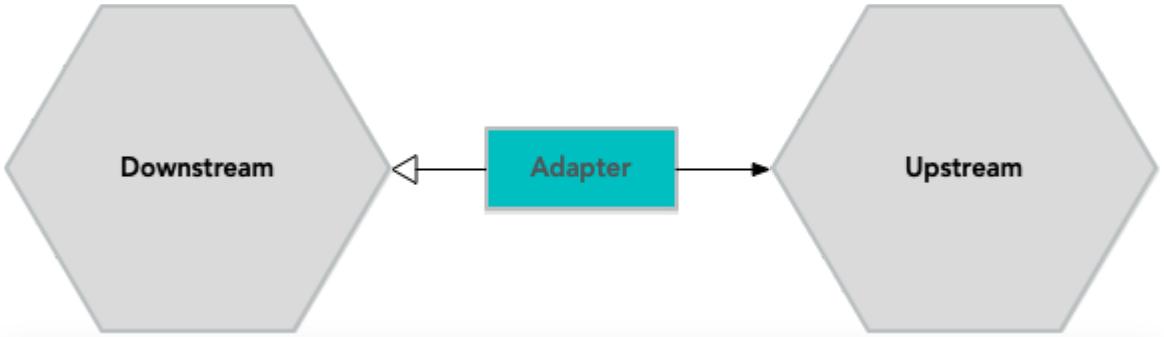
The adapter implements a domain service interface. Like in the previous example, the adapter is injected into the application services using some kind of dependency injection. It then invokes the HTTP client to make calls to the external system and translates the received responses into domain objects.

So far we have only look at what the hexagonal architecture looks like when applied to a single bounded context. But what happens when you have multiple bounded contexts that need to communicate with each other?

If the contexts are running on separate systems and communicating over a network, you can do something like this: Create a REST API for the upstream system and a REST client adapter for the downstream system:



The mapping between the different contexts would take place in the downstream system's adapter.



Since both contexts are running inside the same virtual machine, we only need one adapter that interacts with both contexts directly. The adapter implements an interface of the downstream context and invokes application services of the upstream context. Any context mapping takes place inside the adapter.

Next: Domain-Driven Design and Spring Boot

In the next and final article in this series, we are going to learn how to use Spring Boot to build applications using domain-driven design and the hexagonal architecture.

10

Comments (19)

Please, login or signup to start new discussion.

Данил Прокопчук 1 week ago

Thanks for this excellent article. Many important topics was highlighted on it. One thing which I couldn't find an answer on it is 'Is it acceptable to work with DTO models at the domain services?'. Could you please share your thought on it?

↳ **Petter Holmström** 15 hours ago

I don't see why you would not be able to work with DTOs at the domain service level if there is a good reason for it. Passing UI-specific DTOs down to the domain services is not a good idea, but creating domain-service specific DTOs (for example a Filter object) is fine IMO.

Joaquín Vegara García 20 hours ago

Petter Holmström 15 hours ago

Thanks for the feedback! The corona virus situation has messed up all my schedules so I really can't tell when the next article will be available. I'm going to do a slight revision of this article before that, regarding the ports (see the comments below for more information).

Z Pavic 3 months ago

I don't believe your description of *ports* matches what is intended as part of hexagonal architecture although I'll admit there is some confusion in various articles on the topic. In my understanding, partially enforced by the content on clean architecture, that the ports are the interfaces between the application core and adapters that bridge to the outside world

To quote the original ports-and-adapters article: "As events arrive from the outside world at a port, a technology-specific adapter converts it into a usable procedure call or message and passes it to the application. The application is blissfully ignorant of the nature of the input device." This seems to agree with your diagram and interpretation here.

However, this continues: "When the application has something to send out, it sends it out through a port to an adapter, which creates the appropriate signals needed by the receiving technology (human or automated). The application has a semantically sound interaction with the adapters on all sides of it, without actually knowing the nature of the things on the other side of the adapters." This outlines a different flow, presumably just(?) for the secondary port.

Further on: "The protocol for a port is given by the purpose of the conversation between the two devices. The protocol takes the form of an application program interface (API)." Certainly, something like

The clearest picture may be figure 4 in the seminal article (https://web.archive.org/web/20180822100852/http://alistair.cockburn.us/Hexagonal+architecture). This diagram contains the various devices and external systems outside the hexagon, several adapters listed in the outer ring, the inner ring boundary with ports noted, and the application for the inner (core) hexagon.

In any case, a great article here. Thanks for putting it up!

↳ Petter Holmström 2 months ago

Hello and thanks for your comment.

To be honest, I have not read the original article about ports and adapters. My description is based on my interpretation of how the architecture was presented in "Implementing domain-driven design" and I tried to make the most sense of it in the context of the technologies that we are currently using.

However, I kind of like the idea presented in the article (provided that I understood it correctly) about defining ports by purpose (trigger data, notifications, database, administration etc.) rather than actual input protocol. In that way, the port becomes more of a conceptual thing and the physical "port" (HTTP, JDBC, etc). would essentially be an implementation detail of the different adapters. This would actually make more sense since you would not be able to find the concept of the ports in the code if you were to use my original interpretation of the concept of ports. With the conceptual interpretation, you could use packages for the conceptual ports and put the adapters inside these packages as either classes (simple adapters) or subpackages (complex adapters) and that would make the ports more tangible. This again opens up for some interesting reasoning.

my mindset on the matter and it will be reflected in future articles on the subject.

-Petter-

Andy Bowles 1 month ago

After some reading up on "ports and adapters", I would second Z Pavic's impression that your article mixes up ports and adapters. Considering that this article seems to be read quite a lot, might I suggest that you **do** adapt your text, to avoid confusion for your readers? Thank you, and thank you anyway for the effort you put into this article and the whole series!

Petter Holmström 1 month ago

I am going to address this in some way, but I have not had time to do it yet.

I don't have Vaughn Vernon's book with me at the moment so I can't double check but I'm pretty sure the version of ports and adapters I describe here is also the version he describes in his book (IIRC there was even a section about having separate ports for output and input, so that the adapter was reading data from the input port and writing to the output port). So I have to somehow take both authors' perspectives into account. I don't consider myself to be an expert enough to say that one of them is right and the other one is wrong.

Another option is of course that I just have misunderstood what Vernon wrote, in which case updating the article will be a lot easier.

In any case, I will address this in some way as soon as I have time. I find this subject quite interesting so I'm looking forward to do the necessary research.

I have the impression that it is Vernon who misunderstands Cockburn.

Cockburn (https://web.archive.org/web/20180822100852/http://alistair.cockburn.us/Hexagonal+architecture):

"An adapter maps (different things from the outside) to the API of the port"

"A port identifies a purposeful conversation. There will typically be multiple adapters for any one port, for various technologies that may plug into that port. Typically, these might include a phone answering machine, a human voice, a touch-tone phone, a graphical human interface, a test harness, a batch driver, an http interface, a direct program-to-program interface, a mock (in-memory) database, a real database (perhaps different databases for development, test, and real use)."

From outside to inside: adapter -> port -> application

Vernon in "Implementing Domain-Driven Design" (https://books.google.com/books?id=X7DpD5g3VP8C&pg=PA125), after referring to Cockburn:

"In Figure 4.4 each client type has its own Adapter (Gamma et al.), which transforms input protocols into input that is compatible with the application's API—the inside. Each of the (outer) hexagon's sides represents a different kind of Port, for either input or output. Three of the clients' requests arrive via the same kind of input Port (Adapters A, B, and C)"

"The application boundary, or inner hexagon, is also the use case (or user story) boundary."

From outside to inside: port -> adapter -> application

↳ Petter Holmström 1 month ago

My impression is that for Cockburn, the port is more of an abstract concept. The port is more about the purpose or nature of an interaction with the outside world.

Vernon, again, seems to see ports as physical inputs and output channels such as HTTP. He mentions this as an example: "Think of a Port as HTTP and the Adapter as a Java Servlet or JAX-RS annotated class that receives method invocations from a container...". However, this is clearly not what Cockburn means.

In this way, Vernon's approach - and the one I have in this article - is actually making things more complicated than they need to be. Having the port as a physical input or output does not really bring much value to the design, except maybe from the point of view and deployment and which network ports you need to open. It also does not give you any information about how the different adapters are to interact with the application itself (which services to use, how to use them and so on).

Making the physical port a part of the adapter and the conceptual Port into a "specification of intent" makes this a lot clearer. The UI adapter would interact through the port that is intended for human users, where as the database adapter would interact through the port that is intended for database storage.

What are your comments on this interpretation?

Kamil Olszewski 3 months ago

Wonderful article series, would love to see the next chapter. How is your customer project going? Any chances for grande-finale in upcoming months? :)

Hi Kamil and thanks for your feedback. Unfortunately it looks like I will be busy with the project for at least the following four months so I can't make any promises.

-Petter-

Marco Missfeldt 6 months ago

Hi Petter... thanks for writing these very elaborated articles. What I am really missing and waiting for is the architecture proposal for the user client - until here just painted as a single cloud symbol. From a users perspective I want a user interface that aggregates all business aspects of a process in a comprehensive and seamless way. So the emerging question is - be it with Vaadin or not - how to build a modular but integrated frontend with ui parts coming from many different domain modules. I built such a system many years ago, but the domain modules were deployed as a monolith, so it was possible to show up as a single application against the browser. It would be very interesting how you would do this in such a distributed and event driven system.

↳ Petter Holmström 5 months ago

Hi Marco,

Thank you for your feedback.

Yes, that would indeed be an interesting topic to explore. There are some alternatives to choose from and I have made some small PoC-experiments myself. I have not yet come up with a solution that feels really nice, though.

Unfortunately I'm very busy with a hectic customer project right now so I don't expect there will be any progress on this for many months to come.

Lisandro Martinez 7 months ago

Hi, Petter. Great article. I can't wait to read the next one.

Jeroen Druwé 7 months ago

@Petter, nice article. Looking forward to the next part :)

Yousef Salah 7 months ago

Hi Petter, I'm awaiting your next article too, The crash course is amazing and your writing style is awesome. Keep it up man!

Ishan Soni 7 months ago

Hi Petter, this is an excellent series! Did you get the chance to work on the last article?

↳ Petter Holmström 7 months ago

Hi! I have not yet started to work on the last article but it is on my to-do list. It's nice to hear that there is an audience waiting, though. :-)

-Petter-

Ask questions and get help on the Vaadin Forum.

Follow development, submit issues and patches on GitHub.

Ask for help and chat with project maintainers on Gitter.

BUILD

Get started

Components

Add-on Directory

TOOLS

Designer

TestBench

Multiplatform Runtime

RESOURCES

Labs

Students

LEARN

Vaadin features

Tutorials

Training

Documentation

API

FAQ

INSIGHTS

Progressive Web Apps

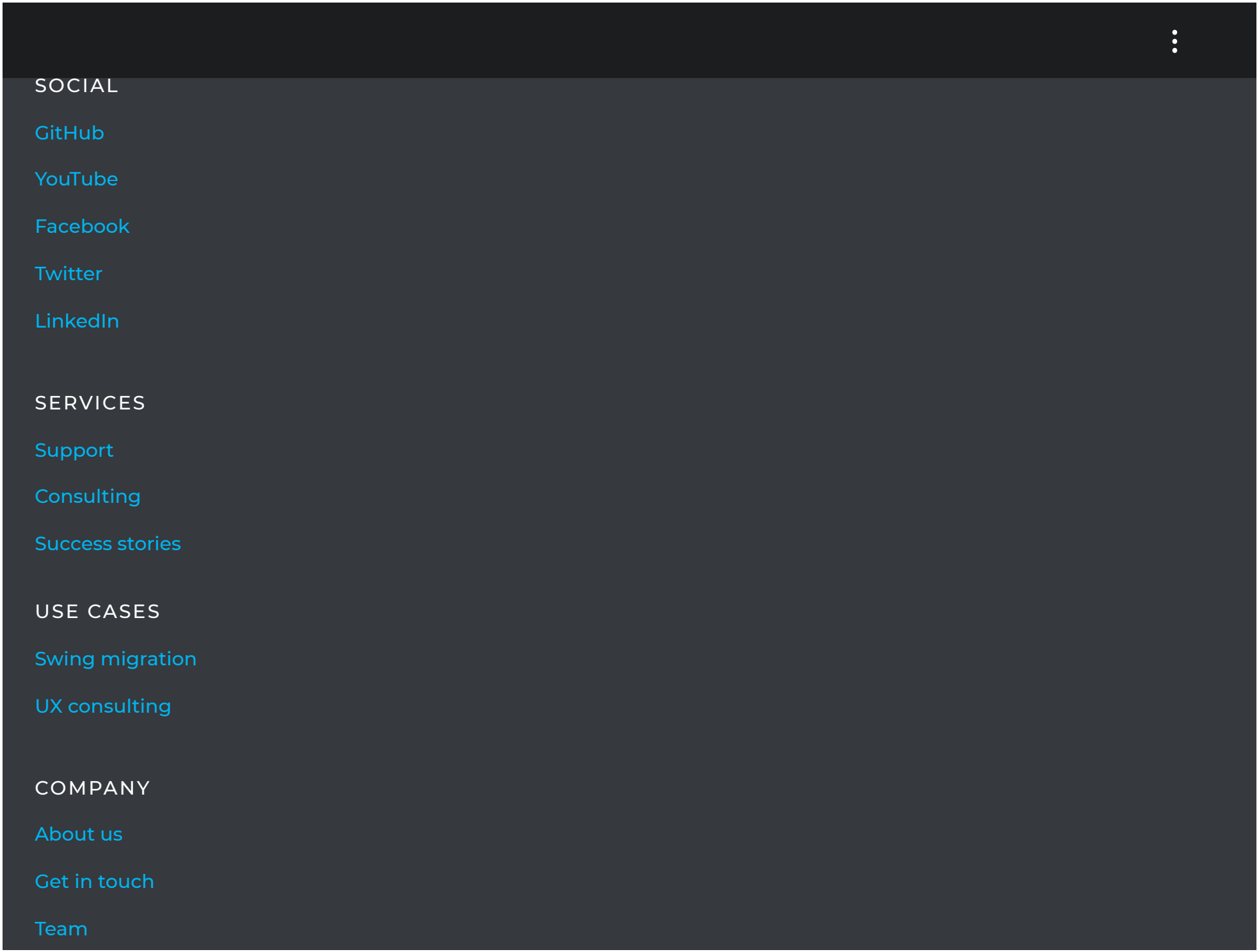
Vaadin 8 LTS

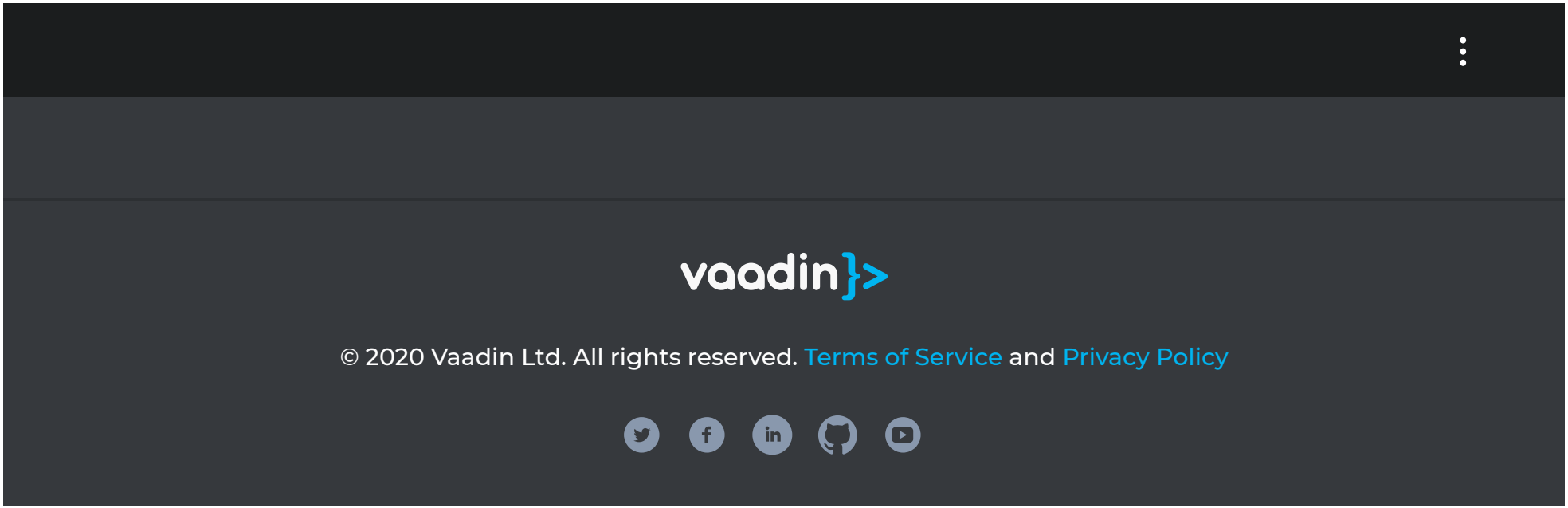
COMMUNITY

Forum

Blog

Add-on Directory


SOCIAL
GitHub
YouTube
Facebook
Twitter
LinkedIn
SERVICES
Support
Consulting
Success stories
USE CASES
Swing migration
UX consulting
COMPANY
About us
Get in touch
Team


vaadin}>
© 2020 Vaadin Ltd. All rights reserved. Terms of Service and Privacy Policy