MULTI-TIERED APPLICATION DEVELOPMENT: FROM LEGACY TO MODERN JAVA EE

OLIVIER LIECHTI

A hands-on approach to design patterns and engineering practices

2018 – v0.4

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ACRONYMS

AOP	Aspect Oriented Programming
API	Application Programming Interface
CLI	Command Line Interface
CRUD	Create Read Update Delete
DAO	Data Access Object
EJB	Enterprise Java Beans
GUI	Graphical User Interface
IoC	Inversion of Control
IIOP	Internet Inter-Orb Protocol
Java EE	Java Enterprise Edition
JAX-RS	Java API for RESTful Web Services
JDBC	Java DataBase Connectivity
JMS	Java Messaging Service

JPA	JPA Persistence API
JSF	Java Server Faces
JSP	Java Server Pages
JSR	Java Specification Request
JSTL	Java Standard Tag Library
JVM	Java Virtual Machine
MDB	Message Driven Bean
MVC	Model View Controller
ORM	Object Relational Mapping
POJO	Plain Old Java Object
RDBMS	Relational DataBase Management System
RMI	Remote Method Invocation
SFSB	StateFul Session Bean
SLSB	StateLess Session Bean
SPA	Single Page Application
SPI	Service Provider API
WAR	Web ARchive

Part I

INTRODUCTION

This introductory part introduces the notion of *multi-tiered software architecture*. It proposes a reference model, which consists of five different tiers. It also gives a high-level overview of the Java Enterprise Edition (Java EE) platform. It explains how this particular technology stack can be used to *build* and *run* multi-tiered applications. The Java EE development cycle, through which developers write Java code and generate Web ARchive (WAR) packages then deployed in a compliant application servers is presented. In terms of practice and experimental tasks, while reading this part, the reader is invited to install and execute three different Java EE application servers. The reader is also invited to deploy an existing application in these servers, to understand what it means to run a Java EE application.

1 FIRST EXPERIMENTS WITH JAVA EE

1.1 INTRODUCTION

The goal of this chapter is to give a first and rapid introduction to the Java EE platform. It is to explain how this set of technologies can be used to build and run multi-tiered applications. We introduce concepts, tools and procedures required for building and executing Java EE applications.

The goal of this course is to study design patterns that apply to any multi-tiered application. We use Java EE to illustrate these patterns with concrete implementations, but the concepts are not specific to this technology stack.

Java EE is only one of the platforms that can be used for building multi-tiered applications. Microsoft .Net is another popular platform based on very similar concepts. There are also related frameworks in Javascript, Python, PHP and other languages.

This chapter introduces a 5-tiers reference architecture and describes the role of every tier at a high level. Every tier will be studied in details in a subsequent chapter. Our goal here is to present the big picture.

In addition to the conceptual overview, we go through concrete operations to get a feel for the developer experience. After reading this chapter, you should have installed at least one of the Java EE application servers, both locally and with Docker. You should also have deployed an existing application, provided to you in the form of an archive file (a `.war` file).

1.2 MULTI-TIERED APPLICATIONS

The term *multi-tiered* refers to a particular *style* of *software architecture*. Some applications follow similar patterns in the way they are structured and in the way they interact with external systems. When this is the case, one can say that they follow the same architectural style. In other words, an architectural style is a collection of design constraints and decisions that are common to a class of applications.

Multi-tiered is a fancy word for describing a distributed application, where a request is issued by a client, received on a server and processed by several components.

Many applications that are used by professionals and consumers have been designed as multi-tiered applications. Here are some aspects they have in common:

- An entity uses a device to interact with the application. Think of a user accessing a web application with a browser. Think of a user interacting with a mobile app on his phone. Think of a user typing a command in a Command Line Interface tool. The entity is often a human, but it can also be a script, a robot, a sensor or or an actuator.

- At some point, the user interaction triggers a call to a *remote* service. This typically happens when the user sends a *query* or a *command* to a remote service. The remote call is made with an application protocol, built on top of a transport protocol. Nowadays, this often means sending a JSON payload over HTTP. But in the early days of Java EE, this often meant making remote method calls from rich clients, with the Java Remote Method Invocation (RMI) protocol.

- The request is processed by a series of components, which collaborate with each other and focus on a particular task. In other chapters, we will examine the server-side Model View Controller pattern. We will also study different ways to interact with database management systems.

- During the processing of a request, business logic is executed and the state of the application is retrieved and/or updated. In the end, a response is sent back to the client tier, so that feedback can be presented to the user.

A multi-tiered application is a *distributed* application. In general, there is a least one *client* machine and one *server* machine, connected with a network. It is also common to use several machines on the *server* side. For instance, there might be a machine that executes the business logic and another machine that stores data. In addition, one can look at the *physical* and *logical* boundaries between tiers. In other words, when looking at two application components that communicate with each other, one should ask if they are located:

Designing the architecture of a system is about making tradeoffs. Always keep in mind that remote calls are extremely expensive and that interprocess call are very expensive, compared to local method calls. Security, scalability, availability can be arguments for a physical separation, but are they relevant in your application?

- on different *machines*
- on the same machine, but in different *processes*
- in the same process (they can still be cleanly isolated in *source code packages*)

Deciding how to split an application and how to use physical and logical tiers can be a bit tricky. It also has a significant impact on non-functional requirements (performance, scalability, security, etc.) and as always, there are pros and cons. In the early days, J2EE guidelines could give the false impression that everything had to be separated. For many simple applications, this translated into higher complexity and lower performance, without clear benefits. Over the years, the community became aware of the design choices and of their implications. Applications developed on top of Java EE became simpler. Logical separation was increasingly preferred to physical separation.

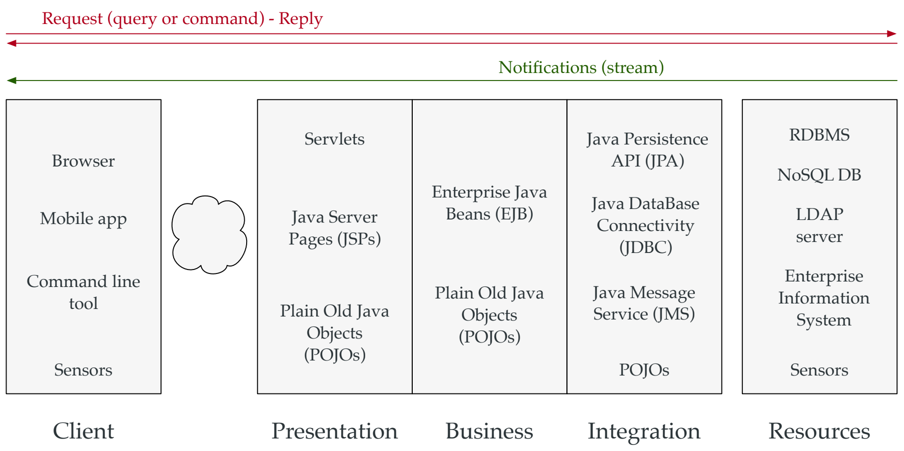

Request (query or command) - Reply
Notifications (stream)
Browser
Mobile app
Command line tool
Sensors
Servlets
Java Server Pages (JSPs)
Plain Old Java Objects (POJOs)
Enterprise Java Beans (EJB)
Plain Old Java Objects (POJOs)
Java Persistence API (JPA)
Java DataBase Connectivity (JDBC)
Java Message Service (JMS)
POJOs
RDBMS
NoSQL DB
LDAP server
Enterprise Information System
Sensors
Client
Presentation
Business
Integration
Resources

Figure 1.1: The five-tiers reference architecture

1.3 THE FIVE-TIERS REFERENCE ARCHITECTURE

The term multi-tiered architecture does not specify how many tiers should be used to organize application components. The client-server architecture is a multi-tiered architecture with 2 tiers. The 3-tiers architecture is a simple model, which emphasizes the separation between the user interface, the business logic and the data persistence. Modern distributed application platforms, such as Java EE or Microsoft .Net often use a more detailed model. These platforms consist of many different APIs and the detailed model is useful to understand where they operate and how they relate to each other. In the case of Java EE, these APIs include the Servlet API, Java Server Pages (JSP), Java API for RESTful Web Services (JAX-RS), Enterprise Java Beans (EJB), JPA Persistence API (JPA), Java Messaging Service (JMS), etc.

A graphical representation of the 5-tiers reference architecture is shown in 1.1. The boundaries between the boxes suggest that, very often:

- the client tier is *far* from the other tiers (they are connected by the Internet)
- the presentation, business and integration tiers are close to each other (they are often located within the same process)
- the presentation, business, integration and resources tiers are often connected by the same internal network (calls from the integration to the resources tier are remote, thus quite expensive, but much less expensive than calls that go over the Internet).

1.3.1 *Client tier*

The client tier represents what I have in my hands.

The client tier consists of the components that are very close to the user. Typically, the user has some sort of device, such as laptop, a mobile phone or a wearable device. This device provides a runtime environment for our application components. Sometimes, it is the operating system (if we write a standalone client application). Very often, it is a higher-level environment, such as a web browser (which renders our markup pages and runs our client-side Javascript functions).

1.3.2 *Presentation tier*

The presentation tier is on the server side. It is responsible for accepting requests and generating views.

The presentation tier is located on the server side, where it provides an entry point. Components in this tier have the responsibility to process incoming requests (e.g. HTTP requests), to delegate work to the business tier and obtain data, and finally to render this data in views that are sent back to the client tier. A common scenario is for the client tier to generate HTML pages, which are then rendered by a browser on the client side. In this case, the page navigation and user interface is managed on the server side. Another common scenario is is for the client tier to generate JSON documents, which are then processed by Javascript components within a browser. In this case, the page navigation and user interface is at least partly managed on the client side. The Single Page Application (SPA) architecture is an example for this scenario.

1.3.3 *Business tier*

The business tier knows nothing about the user interface, nothing about the protocol used to send client requests. It is purely about business logic.

The business tier is decoupled from the user interface and is concerned only with business logic. The same business service can serve requests coming from multiple user interaction channels. For instance, in an online shop application, a *shopping cart* service can be used to serve mobile and desktop users.

1.3.4 *Integration tier*

The integration tier bridges the gap between business logic and data stores.

The integration tier contains components that make the bridge between business logic and data stores. The integration tier provides an abstraction layer, which means that business services do not interact directly with the data stores. Object-Relational Mapping (ORM) middleware is one way to achieve this goal. In enterprise applications, business services do not only interact with databases. They also interact with other applications, either via connectors or via messaging services. The APIs that support these interactions also fit in the integration tier.

1.3.5 *Resource tier*

The resource tier is about data stores, but also about external, often legacy, entreprise applications

The resource tier contains both data stores (databases, directory servers) and external systems that are part of the enterprise information system (e.g. legacy applications). These are the *assets* that the application can use. Are there application components in this tier? It depends on the definition and on the design choices. If all the logic is located in the business tier, then the answer is no. If some of the logic is placed in the data store (e.g. stored procedures in a relational database), then it is yes.

1.4 APPLICATION SERVERS

What does it mean for a developer to create a multi-tiered application? How does one create application components that will live in the different tiers? What does it look like in the code, and how does one make this code executable? To answer these questions, it is necessary to understand the role the the application server, which provides a Java EE *runtime* environment.

To execute Java EE code, we need an application server. When we launch it, it accepts HTTP requests on 2 ports. The first port is used for its admin interface. The other port is used for deployed applications. Applications are deployed in the server. When a request arrives, the server forwards the request to one of the deployed applications.

Before explaining what this means, let us consider what happens with a pure client application built with the Java Standard Edition. Think about a desktop application with a Graphical User Interface or a Command Line Interface. In this case, the code runs directly in the Java Virtual Machine. The user receives a `.jar` file, containing compiled classes and resources, and types a command such as `java -jar application.jar` to execute the code. When the command is executed, the JVM starts, loads the classes and executes the `main` method. The combination of the JVM and of the Java standard library provides a runtime environment: a place where application code can be executed.

With Java EE, the situation is a bit different because the runtime environment offers more APIs and more services. It offers the ability to accept HTTP requests, to interact with databases, to manage security, etc. The runtime environment is provided by an *application server*: a software that implements the Java EE specification (and more) and where applications can be deployed.

As shown in Figure 1.2, the same server can be used to run multiple applications. It accepts requests on a single TCP port and uses the first element in the path to forward the incoming request to the relevant application. Application servers often provide a web-based administration UI, accessible via a different TCP port. This interface is used to deploy applications, to configure connections to databases and to configure all sorts of technical parameters.

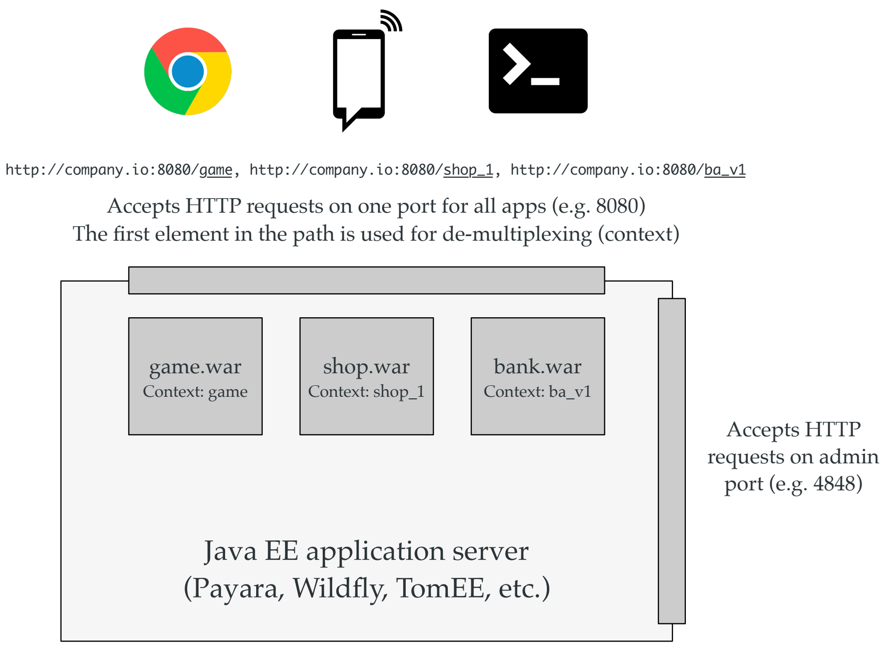

http://company.io:8080/game, http://company.io:8080/shop_1, http://company.io:8080/ba_v1
Accepts HTTP requests on one port for all apps (e.g. 8080)
The first element in the path is used for de-multiplexing (context)
game.war
Context: game
shop.war
Context: shop_1
bank.war
Context: ba_v1
Accepts HTTP requests on admin port (e.g. 4848)
Java EE application server
(Payara, Wildfly, TomEE, etc.)

Figure 1.2: Applications deployed in a Java EE server

1.4.1 *Java EE, the Java Community Process and Java Specification Requests*

There is not only one company driving the specification of Java EE. There is not only one company providing the implementation of the specification.

A JSR is similar to an RFC: it is the specification of an open standard. Companies and open source communities can then build compatible (and competing) implementations.

Java Enterprise Edition was designed as a standard platform, through the *Java Community Process*. This means that expert groups, with representatives from different companies, have first written the specifications for the different aspects of the platform. The term *Java Specification Request* (JSR) refers to one such specification: there are JSRs related to presentation-tier technologies (e.g. Servlet, JSP, JAX-RS), there are JSRs related to business-tier concerns (e.g. EJB), etc. Note that the JCP is not only used for Java EE APIs, but for the Java platform in general.

The standardization process means that when a JSR has been defined, different companies and open source communities can provide a competing implementation of the specification. This means that when you decide to use a particular JSR, you also need to decide which implementation you want to use.

1.4.2 *Choosing an application server*

In the early 2000s, most application servers were commercial servers sold by companies like IBM, BEA and Sun Microsystems. They were expensive and were targeting enterprise customers. In the mid 2000s, open source implementations became viable alternatives. Today, many enterprise customers have moved away from commercial implementations and use open source servers.

The following list is not exhaustive, but gives an overview of available servers:

- *Wildfly*: an open source application server developed by Red Hat. Open source, free, leading edge.
- *JBoss Enterprise Application Platform*, also provided by Red Hat. Commercial support, certified.
- *Glassfish*: initially developed by Sun Microsystems, former reference implementation. Used to work very well, but was killed by Oracle.
- *Payara*: fork of the Glassfish codebase, developed by the Payara Foundation. Addresses issues with last versions of Glassfish, aims to become a leader in the future Java EE versions.
- *Apache Tomcat*: strictly speaking, not a Java EE application server because it only implements a subset of the Java EE APIs. Extremely popular and very often used to deploy applications built with the Spring Framework.
- *Apache TomEE*: the combination of Tomcat and OpenEJB, OpenJPA, MyFaces to fill the gap between Tomcat and the full Java EE specification.
- *WebLogic*: previously developed by BEA, then acquired by Oracle. Used to be an expensive and popular solution.
- *Websphere*: developed by IBM. Price wise, in the same category as WebLogic.

To get familiar with Java EE development, it is worth experimenting with 2 or 3 different application servers. This helps understand what is defined in the specification and common to all application servers. This also helps realize that some aspects are not defined in the specifications (e.g. what configuration interface should be provided), and that every application server is free to provide additional features.

Suggestion: experiment with Wildfly, Payara and TomEE. Make sure that you understand how to deploy a .war file in each of the Java EE application servers. Pick your favorite and dig deeper in the configuration.

As of today, we recommend to experiment with Payara, Wildfly and Tomcat/TomEE. Depending on your preferences, we then recommend that pick one of them and dig into the details of the administration and configuration interfaces.

1.4.3 *Installing and running an application server*

There are two ways to get one of these application servers up and running:

A demonstration of the installation process is available in the Webcast. Be aware that it was recorded with previous versions of the application servers.

- Today, there is an official Docker image for every open source server. The documentation of the images give instructions for running a container and performing basic configuration (in many cases, it is necessary to create user accounts to get access to the administration console). The great thing about this method is that you can get started within minutes. The other great thing is that you have a way to package and distribute your applications, by creating your own Docker images.

- For developers, it is sometimes better to have a local installation of the application server. This enables some features in the IDE and makes the development cycle more efficient. The three open source application servers are very easy to install: the process essentially consists of extracting an archive file and running a script in a bin directory.

1.5 THE JAVA EE DEVELOPMENT CYCLE

Having silos in the organization and a clear separation between development and IT operations used to be the norm. This has proven to cause many problems. The trend is to break these barriers and to work in small autonomous teams. This is a core principle in the DevOps approach.

The original Java 2 Enterprise Edition specification described a complete software development lifecycle, with clearly defined engineering roles. In the early days, it made the boundaries between *building* software and *running* software very explicit. This reflected the organization of most companies at the time, who had separate *development* and *IT operations* teams. In this model:

- software engineers were responsible for building components and applications. Ideally, they should be able to focus on business logic and not worry (too much) about technical challenges. In theory, they should be able to *declare* the non-functional requirements of their applications (availability, security, etc) and let the runtime environment take care of the rest.
- IT ops engineers were responsible for running these applications. They were responsible for designing (often distributed) runtime environments, for deploying applications and for making sure that they kept working. IT ops were the engineers carrying pagers and dealing with production issues.

A .war file is almost the same thing as a .jar file. It is an archive, which contains compiled classes, resources and metadata. The structure of folders and the name of files is defined in the specification.

In this model, how do software and IT ops engineers collaborate? What happens when a new feature has been developed, or when a bug has been fixed in the source code? Depending on the environment and the type of application, this can be a complex process. One aspect of this process is to have a way to *package* enterprise application in some sort of archive, which can be produced by software engineers and delivered to IT ops engineers.

When Java EE was created, the Java platform had already defined a format for creating packages: the *Java ARchive* format. Java developers are familiar with *.jar* files, which contain compressed .class files, resources and metadata information. These files are used to distribute libraries and applications executed directly in the Java Virtual Machine. The same approach was used for Java EE and several related archive formats have been proposed:

- The *WAR* format was proposed for web applications. In the early days, the intent was to package *presentation-tier* components in these archives. In other words, the components that process HTTP requests, obtain some data and generate views that are sent in HTTP responses. As we will see later, this meant Servlets, Java Server Pages (JSPs) and helper Plain Old Java Object (POJO) classes.

- The *JAR* format was proposed for business components. The idea was to design components that would be decoupled from the presentation tier and could be reused across different user interface channels. These components would need to interact with databases, legacy applications and messaging platforms. They would often require a way to deal with distributed transactions. We we will see, Enterprise Java Beans (EJBs) have been developed for this purpose. EJBs are packaged in .jar files.

- The *EAR* format was proposed for entire applications. For many years, the way to build Java EE applications was to create separate packages for the web tier (in a .war file) and the business tier (in one or more .jar files). All these components were then assembled in a larger .ear file, which contained extra metadata.

It is important to be aware of the original *.ear* packaging model for Java EE applications, because there are still many legacy applications that are built on this model. For a Java developer, there is still a high probability to encounter such an application. But it is also important to understand that over the years, the original model has been revised several times, with the goal to provide a more lightweight experience to developers. For many years, most teams have dropped the EAR format and only use the WAR format. Indeed, it is not possible to put Enterprise Java Beans (EJBs) together with the presentation tiers. In addition, many developers have made the choice not to use EJBs and develop business services with POJOs and the help of frameworks such as the Spring Framework.

In the past, developers had to create .ear files, which contained .war and .jar files. The These archives reflected the tiers of the application. Today, it is possible and more common to put all application components in a simpler .war file. Separation between tiers is visible in the source code package structure.

1.6 ONLINE RESOURCES

The following resources are available for this chapter:

- In the companion YouTube playlist, the sequence of four videos entitled *Bootcamp 1.1* to *Bootcamp 1.4* present the Java EE development workflow and demonstrate how to deploy the same .war file in three different application servers. The videos were recorded in 2016, so be aware that the versions of the servers are outdated. Some of the operations might be slightly different. Also, instead of using Glassfish, we recommend that you switch to Payara.

- On GitHub, the repo SoftEng-HEIGVD/Teaching-HEIGVD-AMT-Discovery contains the `.war` file used in the webcasts. You need to clone in order to go through the tutorial steps. The repo also contains Docker configuration.

1.7 QUESTIONS

To answer these questions, you will need to have read the chapter but also to have done some research. Make sure that you are able to answer every question. Discuss your responses with your peers.

1. *Is Apache Tomcat an application server?* The answer to this question depends on the definition of *application server*. Explain why it is correct to answer positively and negatively to the question.

2. *What is the role of a .war file?* Explain how it fits in the Java EE development model and bridges the gap between development and operations.

3. *Is the J2EE development model an anti-pattern?* On paper, the clear separation between development and operations teams makes sense. However, the current trends in the industry are very different. Explain how modern practices give a different perspective on the question.

4. What is the difference between the *client* tier and the *presentation* tier. They both seem to be related to the user interface, so why are they both needed?

5. Consider the web site `https://qoqa.ch`. It is reasonable to think that it is built on a multi-tiered architecture. Use the 5-layers diagram and draw the components that might be part of the system.

6. We have stated that components in the business tier should be decoupled from the user interface. Use the 5-layers diagram and describe a scenario that shows how a business service can be reused across interaction channels.

7. Most application servers provide 3 ways to deploy a `.war` file. Describe how these methods work and what are there benefits and drawbacks.

Part II

OLD-STYLE JAVA EE

This part introduces the core APIs that are part of the umbrella Java EE specification. We use the terms *legacy* and *old-style* because many new projects use frameworks built on top of the Java EE APIs. These frameworks hide details and offer a higher level of abstraction. Even if they make developers more productive, it is important to understand what happens behind the scenes, in the foundations of the platform. Furthermore, there are still many companies who need to maintain older applications, built directly on top of Java EE. There is a high probability for a software engineer to encounter such applications in the professional life.

In the chapter dedicated to the *client tier*, we review several types of clients (thin clients, rich clients, etc.). We also look at the communication between client-side and server-side components.

In the chapter dedicated to the *presentation tier*, we present three design patterns: the Model View Controller (MVC) pattern, the Inversion of Control (IoC) pattern and the Pipes and filters pattern. We present these patterns in generic terms, as they can be applied in any technology stack. We also show how they can be concretely applied with Java EE components.

[More content to be added for the business, integration and resources tiers.]

2 THE CLIENT TIER

2.1 INTRODUCTION

The client tier is very close to the user. Typically, it consists of components executed on a device that provides some sort of user interface. The user interface might be a Graphical User Interface (GUI) or a Command Line Interface (CLI). There are many different ways to design the client tier, which are reviewed in this chapter.

2.2 THIN WEB CLIENTS

A very common situation is when users interact with the application via a web browser. They launch the browser, type in a URL and get to a user interface that has been generated by the server-side components of the application. The term *thin* means that there is very little logic, and therefore code, running on the server side. Some applications are really, really thin: the browser receives HTML, stylesheets and media files and simply renders them. The application does not use any client-side Javascript, so there is no code executed in the browser. But many applications add a bit of meat around that structure. Some Javascript code is added and executed in the browser, for instance to validate data entered in forms.

If it is really, really thin, the client does not execute any logic. Views generated by the server are simply rendered by the browser. Every user action triggers an HTTP request and a full page reload.

2.3 RICH WEB CLIENT CLIENTS

When the amount of Javascript code grows, the web clients become *richer*. Think about social network applications, online games, or complex business applications: a lot of them now have a very interactive user interface, which requires code to be executed in the browser. The term Single Page Application (SPA) refers to a pattern that has become very popular, and that is enabled by Javascript frameworks such as Angular, React or Vue. Here, most of the user interaction is handled on the client side, including page navigation. At the beginning of the session, the user visits a URL and fetches a skeleton that bootstraps the application. He sees the entry page in the application and start interacting with it. As he clicks on links and buttons, the entire page is never reloaded in the browser. Instead, the browser fetches HTML fragments and application data. It injects these elements in the page skeleton and sometimes gives the impression that a page has been reloaded. Be aware that even if the URL changes in the navigation bar, it is possible that a full page has not been requested by the client.

Rich indicates that there is more interactivity in the user interface, which mandates for some of the UI logic to be running on the client. The logic is implemented in Javascript functions, executed by the engine embedded in the browser.

2.4 NATIVE GUI CLIENTS

In the past, many things were not possible in a web browser. This is the reason why some applications provided a rich desktop client. Today, browsers provide a rich environment and a viable alternative most of the time. Mobile devices are still an area where native clients have benefits.

On the desktop, there used to be a time where many business applications did not use a browser on the client side. Instead, they used a rich client developed with a GUI toolkit such as Swing. One reason for doing that was the need for a high degree of interactivity, at a time where browser capabilities where lacking (we are talking about the early 2000s). Today, there are still some scenarios where a rich desktop client makes sense, but they are not the majority.

However, there is another reason why a lot of multi-tiered systems include a native client: the support for mobile users. In this case, many developers make the choice to implement a native application, for instance on iOS or Android.

When the application includes a native client, one question that needs to be answered is how the client-side components communicate with the server-side components. Today, the most common protocol used for that purpose is HTTP. In this case, HTTP is used to transport data (e.g. JSON, XML) and not HTML documents. In the past, many companies used to build rich clients in Java and use the Remote Method Invocation (RMI) protocol to to handle communications between objects on the client and the server side. This is not a scenario that we will look in details in this book.

2.5 CLI CLIENTS

When you type a command in a terminal, you might well be interacting with the client component of a multi-tiered application

Many applications provide a text-based interface, sometimes in addition to a graphical user interface. One of the benefits of such an interface is that it facilitates scripting and automation. When using a CLI, one might sometimes have the impression that everything runs locally. This is not correct: many programs that implement a CLI are network clients and make remote calls to a service. In other words, when the user types a command on the terminal, the program might send an HTTP request to a remote service, process its reply and format a message on the terminal.

Many applications are designed with the idea that a user will drive the interaction and trigger the execution of business logic. But there are also a lot of applications where the entity initiating the work is a software agent. Think about automation scripts executed periodically. Think about IoT sensors and actuators that send data to a cloud application.

2.6 COMMUNICATION PROTOCOLS

The previous paragraphs have already mentioned some of the communication protocols used between the client tier and the server tiers.

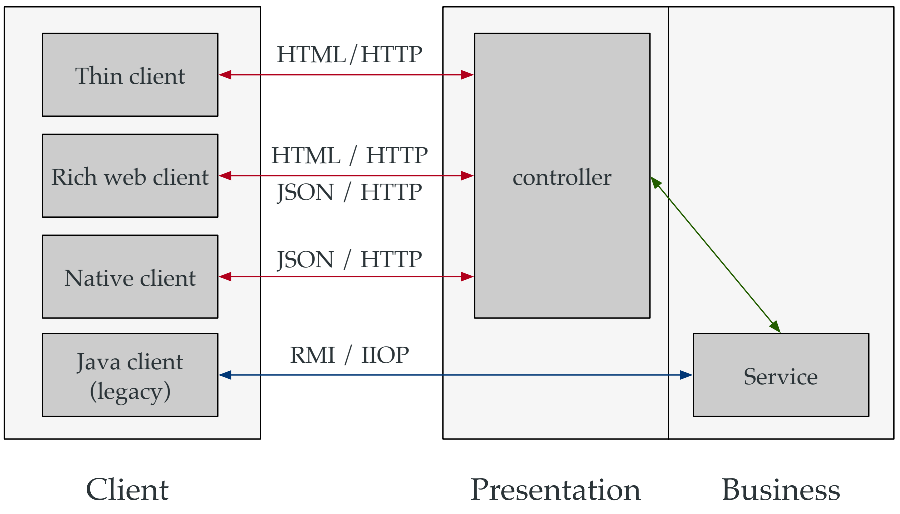


Figure 2.1: Interactions between the client and server tiers

These protocols are shown in Figure 2.1. Here are few comments about every scenario:

- For thin clients, HTTP is used to transport HTML documents (and other web assets like stylesheets and images). Since there is no logic executed on the client, there is no point to send structured business data from the server to the client.
- For rich web clients, HTTP is used to transport UI assets (HTML, stylesheets, media files) but also business data. This data is requested by scripts running in the browser, formatted and injected in the page UI. The figure mentions JSON as a format for serializing business data, but there are other formats (XML, CSV, binary formats, etc.).
- For native clients, HTTP is also often used as the underlying transport protocol. But in this case, there is no web page, so what is transported is structured business data.
- Finally, the case of legacy Java clients is a bit particular. In this case, the objects created in the Java Virtual Machine (JVM) on the client may use the RMI protocol over Internet Inter-Orb Protocol (IIOP) to make method calls to objects created in the JVM on the server side. In this case, the figure suggests that the call can be done directly to the business tier.

2.7 QUESTIONS

To answer these questions, you will need to have read the chapter but also to have done some research. Make sure that you are able to answer every question. Discuss your responses with your peers.

1. Let us consider Facebook. What kind of clients does the company use? Do not forget that a company always has external and internal applications.

2. What does the acronym AJAX mean and how does it relate to the client tier?

3. Give an example of a CLI tool that you use and that is the client of a multi-tiered application.

3 THE PRESENTATION TIER

3.1 INTRODUCTION

The presentation tier is located at the boundaries of the server side: it represents an entry point into the server. Its core responsibilities are to accept requests from clients, to figure out what business logic should be executed and finally to send responses in the appropriate format.

The details of what happens in the presentation tier depends on type of client that sends the request: a thin client requesting web pages and a rich client requesting business data may involve different components on the server side. However, there are design patterns that apply in every situation. The objective of this chapter is to explore them. We will review the MVC, the IoC and the Pipes and Filters design patterns. After presenting them in generic terms, we will see how to concretely apply them with Java EE APIs and look at some code.

3.2 THE MODEL VIEW CONTROLLER PATTERN

MVC is one of the well-known design patterns and has been documented decades ago. It has its origin in the development of graphical user interfaces for desktop applications. The goal of the pattern is to clearly separate responsibilities between three software components. This leads to code that is easier to write, understand and maintain. In many software engineering courses and textbooks, the implementation of MVC is illustrated with GUI toolkits like Swing or Java FX.

MVC was first proposed in the context of GUI applications. In multi-tiered applications, MVC can mean different things. Rich clients and modern Javascript frameworks apply the traditional definition of the pattern. In the presentation tier, the pattern is applied quite differently. It is sometimes referred to as MVC2

In the context of multi-tiered applications, be aware that MVC can take different shapes and may be applied in different tiers. In the case of rich clients, MVC takes its original form and is purely in the user interface domain. If you decide to use a client-side Javascript framework, such as Angular or React, you will be exposed to client-side MVC. In the case of thin clients, MVC is implemented on the server side, in the presentation tier. This changes quite a few things as the implementation does not involve a UI toolkit. In this chapter, we only consider the second situation. In other words, we describe a variation of the MVC design pattern that operates on the server side. The key question is then to understand the relationship between the pattern components and the HTTP request-reply model.

3.2.1 *Description of the pattern*

The pattern involves the following participants:

- the *model* is an object, or a graph of objects, that the user is interested in. Let's imagine that the user is accessing a product catalog: the model would be a List of Product objects, with properties such as Price and Description and with linked objects such as a List of Reviews or a Photo. Always keep in mind that you are manipulating graphs of objects: they need memory and bandwidth.

- the *view* is a component that produces a representation of the model. In the previous example, it could be a component that generates an HTML page with embedded images and hyperlinks. The view is generated on the server side, transferred to the client side, where it is finally rendered by a browser. In some cases, the representation is meant to be consumed by a human (e.g. HTML page or PNG image). In other cases, it is meant to be processed by a software agent (e.g. and XML or JSON payload).

- the *controller* is a component that reacts to user actions, is exposed to the details of the HTTP protocol and coordinates the work between the other components. In the case of thin web applications, whenever the user is taking an action (log in, access a product page, add a product in the shopping cart, send a message), he clicks on a link or a button, which causes the browser to send an HTTP request to the server. Every HTTP request is an event that represents a user action. It is comparable to a mouse event or a key event in a UI toolkit. When the server receives an HTTP request, it delegates its processing to the appropriate controller. The controller then has to extract information from the request (from the URL, the query string and/or the headers). The controller then decides how to generate a model and to ask a view to generate a representation of this model. Finally, the controller has to send the representation back to the client. At this stage, it is once again exposed to the details of the HTTP protocol.

- the *service* is a fourth component that is not strictly mandatory, but that is used very often to keep the code of the controllers small. In theory, a controller could create the model itself: by doing some sort of computation, by looking up information in a data store, etc. In practice, the controller will very often delegate this task to a service. In other words, it will handle the situation like this: "Based on what I have seen in the URL and query string, I decide that Service A can handle the user action and generate a model for me. I also decide that View 1 can generate a representation of the model. I am going to coordinate the work between these two helpers."

Figure 2.1 shows the sequence of events for a typical user interaction round trip.

1. The user is visiting an online shop and is on a page that displays a list of products. For each product, there is a picture, a description, a price and an hyperlink entitled "Show details". The user has found an interesting product and clicks on the hyperlink. This where we start our exploration of the MVC pattern.

2. The browser sends an HTTP request to the server. The method is `GET` and the URL is something like `http://shop.com/productsController?productId=4224&action=showDetails`. You can think of the HTTP request as an event, just like you would have a mouse event or a button even in a GUI toolkit.

3. On the server side, the HTTP request is routed to a component, which is the Controller in our pattern. Depending on the language and platform, the Controller might be an object or a function.

4. The Controller is lazy and tries to do as little as possible. Its first job is to figure out what other components can help him. Firstly, by looking at the attributes of the HTTP request (the URL, the query string, etc.), it will decide which business service can handle the request. It then calls the business service.

5. The Service does its job, which usually means getting some data, applying some business rules and computing a response. The response from the Service to the Controller is the Model in the pattern. It is usually an object or a graph of objects.

6. When the controller has received the Model, it looks for another helper: the View. Again, the controller is responsible to decide which of the available views can generate a representation of the Model. Usually, it looks at the HTTP headers sent by the client: the `Accept` header might indicate if the client prefers HTML or JSON. The Controller must have a way to pass the Model to the View, and to has the View to handle the last part of the processing. This depends on the platform and we will see the Java EE approach later in this chapter, with some code examples.

7. The last step is for the View to do the rendering and to prepare the response, which sent to the client.

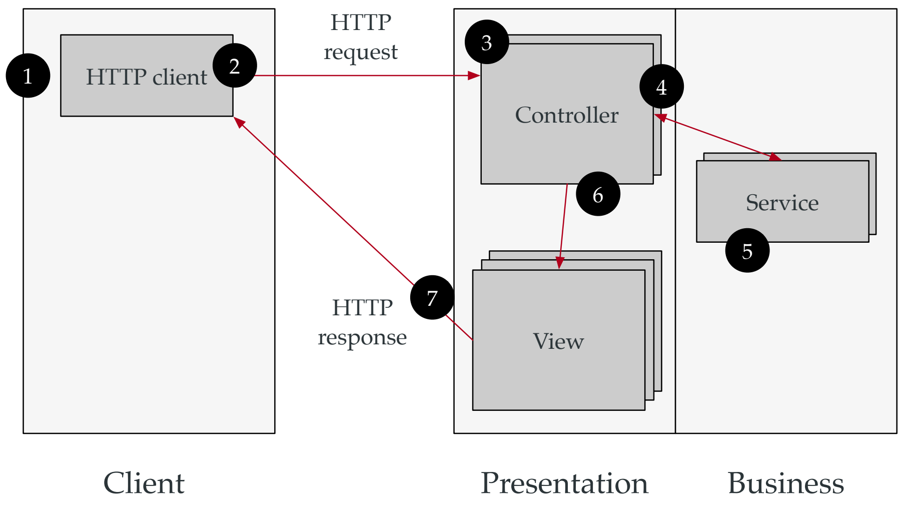


Figure 3.1: The MVC design pattern in the presentation tier

3.2.2 *Applying MVC in Java EE*

Let us know see how we can concretely implement the MVC pattern with Java EE. For that purpose, we introduce two Java Specification Request (JSR): the Servlet API and the JSP API.

A servlet is a Java object that implements a few methods, which are invoked when an HTTP request has been sent by a client. The responsibility of the servlet is to analyze the request (it has access to the URI, the HTTP headers, the body, etc.) and to prepare a response. A simple example is provided in Listing 3.1.

```
1 @WebServlet("/quote")
2 public class QuoteServlet extends HttpServlet {
3   public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws IOException {
4     response.getWriter().println("<html>In cauda venenum.</html>");
5   }
6 }
```

Listing 3.1: A simple HTTP servlet

Let us review the code line by line:

- On line 1, the `@WebServlet` annotation is an instruction given to the application server. It is a way for the developer to state that when an incoming HTTP request targets the `/quote` URL, then the application server should forward it to this servlet. As we will see later, using this annotation is an alternative to the `web.xml` deployment descriptor.

- On line 2, the developer creates a class by extending the `HttpServlet` abstract class, which is defined in the Servlet specification. The specification includes a Javadoc documentation for this class and its methods.

- On line 3, the developer implements a method that will be invoked by the application server when the method in the incoming HTTP request is *GET* (as opposed to *POST*, *PUT*, *DELETE*, *PATCH*, etc.).

- The `doGet` method has two parameters. The `request` parameter is an object that represents a complete HTTP request. The `HttpServletRequest` class is also defined in the Servlet API specification. It provides getter methods to access the request method, URI, headers, etc. The `response` parameter represents the HTTP response. The developer uses this object to prepare the response sent to the client.

- On line 4, the developer obtains a `PrintWriter` object and produces HTML markup that will be sent in the HTTP response body. The developer might also have added headers to the request. He might also have used an `OuputStream` object to send binary data to the client.

This is the simplest example to demonstrate how to build dynamic web applications in Java. But be aware that this example does not follow the MVC pattern. The servlet is at the same time the Controller, the View and the Service (and for this reason, there is no real Model). Mixing HTML markup with Java code is a code smell. It is not rare to observe this in legacy applications and when it is the case, the maintenance is generally very painful.

Behind the scenes, the application server translates every JSP template into Java source code, which it then compiles. The source code is actually a subclass of `HttpServlet`.

Let us improve the code and introduce a View. For that purpose, we will use the JSP technology. Historically, JSP was created a couple of years after the Servlet API. The technology provided a standard way to implement dynamic web pages and has been used extensively. It has evolved a lot over the years and many features have bean added either directly in its specification, or in companion JSRs. Today, JSP is still supported by Java EE application servers and is a perfectly viable choice for building web applications. However, the technology has disappeared from official Java EE tutorials and seems to have lost traction with open source communities. Two factors can explain the situation:

JSP has disappeared from the official Java EE tutorial. The last version that presents the technology and gives example is the Java EE 5 Tutorial.

- In 2004, the Java Server Faces (JSF) API was proposed as an alternative to JSP. The goal was to provide a solution for building component-based user interfaces, on the server side. This was a great idea, but it proved very hard to do. From our own experience, we can say that building web applications with JSF was complex and time consuming. We moved away from the technology before it reached version 2.0, so things might have changed. Even if that is the case, we rarely see new projects adopt this technology, because the trend is now to build rich web clients. Hence, it is the surprising that the official Java EE guidelines put a very strong emphasis on JSF and have even stopped presenting JSP, a simpler and practical solution. Of course, if JSF was widely adopted, it would be benefit certified Java EE application server vendors. The reason is that providing a JSF implementation is mandatory for a fully compliant application server, but it is not for projects like Apache Tomcat.

- At the same time, non-standard alternatives to JSP have been provided by open source projects. JSP is essentially a templating technology, and so are projects such as Thymeleaf, FreeMarker or Velocity. Frameworks that have been developed on top of Java EE, and in particular the very popular Spring MVC, do support JSP, but often start by presenting other template engines.

Let us now how Java Server Pages look like and how they can be used in combination with servlets. The repository SoftEng-HEIGVD/Teaching-HEIGVD-AMT-MVC-simple-example, hosted in GitHub, contains the code that we review here. Instructions for running the code with IntelliJ IDEA are provided in the repository `README.md` file.

```
1  src/main
2  src/main/webapp
3  src/main/webapp/index.jsp
4  src/main/webapp/WEB-INF
5  src/main/webapp/WEB-INF/web.xml
6  src/main/webapp/WEB-INF/pages
7  src/main/webapp/WEB-INF/pages/view.jsp
8  src/main/java
9  src/main/java/ch
10 src/main/java/ch/heigvd
11 src/main/java/ch/heigvd/amt
12 src/main/java/ch/heigvd/amt/mvcsimple
13 src/main/java/ch/heigvd/amt/mvcsimple/business
14 src/main/java/ch/heigvd/amt/mvcsimple/business/QuoteGenerator.java
15 src/main/java/ch/heigvd/amt/mvcsimple/model
16 src/main/java/ch/heigvd/amt/mvcsimple/model/Quote.java
17 src/main/java/ch/heigvd/amt/mvcsimple/presentation
18 src/main/java/ch/heigvd/amt/mvcsimple/presentation/QuoteServlet.java
```

Listing 3.2: Structure of the simple MVC project

The structure of the project source files is shown in Listing 3.2. Here is a description of the different packages and files:

- line 12: we use a namespace for the project: `ch.heivd.amt.mvcsimple`.
- lines 17, 13, 6 and 15: we organize our classes based on the tiers of the architecture. We have controllers in the `presentation` package and services in the `business` package. We have views in the `pages` directory. The `model` package contains the classes that encapsulate business data and are used across tiers.
- line 16: the Quote class is the *model* in the application.
- line 7: the view.jsp JSP is the *view* in the application.

- line 18: the `QuoteServlet` class is the *controller* in the application.
- line 4: `WEB-INF` is a special directory and is present in every `.war` file.
- line 5: `web.xml` is a *deployment descriptor*. It describes the web application and is used by the application server at deployment time. In older versions of Java EE, the deployment descriptor was mandatory for all applications. Over time, the same information can be given to the application server via *annotations* in the code. *Warning*: the `web.xml` indicates the version of the Servlet API used by the application. Using an old version (e.g. by copying old examples found on the Web) may cause hard to debug problems.
- line 6: the pages directory is located inside the `WEB-INF` directory. This is a security best practices. The Servlet specification states that it is not possible to access resources directly (i.e. it is not possible to type a URL in the browser to access them). The only way to reach them is to go via a controller.

Now that we have seen the structure of the project, let us look at the code of the model (Listing3.3), the controller (Listing3.4) and the view (Listing3.5).

```
 1  package ch.heigvd.amt.mvcsimple.model;
 2
 3  public class Quote {
 4
 5      private String author;
 6      private String citation;
 7
 8      public Quote(String author, String citation) {
 9          this.author = author;
10          this.citation = citation;
11      }
12
13      public String getAuthor() {
14          return author;
15      }
16
17      public String getCitation() {
18          return citation;
19      }
20
21  }
```

Listing 3.3: The model class

Here are some comments about the `Quote` model class:

- lines 13 and 17: these two methods are often called getters, because they are used to get the value of an object property. The convention is to name these methods `getXXX()`, where `XXX` is the name of the property. Using this convention is not only a good practice that makes the code uniform and easy to read. It enables some of the magic that happens behind the scenes, because it facilitates dynamic code invocation. We will see a first simple example when we get to the code of the JSP tempate. When we present the notion of Object Relational Mapping (ORM) later in the book, we will also explore the underlying mechanisms in more details.

- line 8: observe that the constructor allows us to define the state of the model object and that we did not implement any setter method. This is a good practice, when we want to work with immutable objects.

```
1  package ch.heigvd.amt.mvcsimple.presentation;
2
3  import ch.heigvd.amt.mvcsimple.business.QuoteGenerator;
4  import ch.heigvd.amt.mvcsimple.model.Quote;
5
6  import javax.servlet.ServletConfig;
7  import javax.servlet.ServletException;
8  import java.io.IOException;
9  import java.util.List;
10
11 public class QuoteServlet extends javax.servlet.http.HttpServlet {
12
13     private QuoteGenerator service; // we will see later how to replace this with
            dependency injection
14
15     @Override
16     public void init(ServletConfig config) throws ServletException {
17         super.init(config);
18         service = new QuoteGenerator();
19     }
20
21     protected void doGet(javax.servlet.http.HttpServletRequest request, javax.
            servlet.http.HttpServletResponse response) throws javax.servlet.
            ServletException, IOException {
22         List<Quote> model = service.generateQuotes();
23         request.setAttribute("quotes", model);
24         request.getRequestDispatcher("/WEB-INF/pages/view.jsp").forward(request,
               response);
25     }
26 }
```

Listing 3.4: The controller class

Here are some comments about the `QuoteServlet` controller class:

- line 11: we implement a servlet, which extends the abstract `HttpServlet` class.

- line 13: we use service, which we instantiate ourselves for the moment, when the servlet is created by the application server (line 18)

- line 21: we implement the `doGet` method, which receives a request and a response in parameter. In the method, we invoke the service and obtain a model (a list of quotes). With `request.setAttribute`, we attach the model to the request. This will allow other components, and in particular the JSP template, to retrieve the model down the processing chain.

- line 24: the way to pass the request to the view is to obtain a request dispatcher for the target template and to call the forward method.

```
 1 <%@ page contentType="text/html;charset=UTF-8" language="java" %>
 2 <%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
 3 <html>
 4   <head>
 5     <title>Quotes</title>
 6   </head>
 7   <body>
 8     <h2>Quotes</h2>
 9     <ul>
10       <c:forEach items="${quotes}" var="quote">
11         <li>${quote.author} : "${quote.citation}"</li>
12       </c:forEach>
13     </ul>
14   </body>
15 </html>
```

Listing 3.5: The view template

Here are some comments about the `page.jsp` view template:

- line 2: in this example, we use the Java Standard Tag Library (JSTL) tag library. Tag libraries are a mechanism for defining custom tags, which can be used in page templates. Here, we specify that JSTL tags will be prefixed by `c:` (line 10). Warning: a fully compliant Java EE application server has to provide an implementation of JSTL. Tomcat, however, is not fully compliant. Therefore, if you deploy the application in Tomcat, you will need to bundle the JSTL `.jar` file. When using maven, the `scope` of a dependency defines whether the dependency is bundled in the `.war` file or if it is `provided` by the application server runtime environment.

- line 10: the `c:forEach` tag allows us to iterate over a collection. In this case, `${quotes}` means that the engine will try to find a model named `quote` in different scopes (page, request, session, application). Remember that in the servlet, we used the method `setAttribute('quotes', model)`, which is why the template can retrieve the model.

- line 11: in the loop, the `quote` variable represents one quote in the list. With the expressions `${quote.author}` and `${quote.citation}`, we can retrieve the properties defined in the model class `Quote`). How does that work? Remember that we talked about naming conventions and getter methods. Behind the scenes, when the template engine sees `quote.citation`, it computes the name of a method (`get` + capitalized name of the property) and attempts to make a dynamic call to this method. The Java mechanism that makes this possible is the Java Reflection API.

```
1  <web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
2          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3          xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
4                  http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd"
5          version="3.1">
6    <display-name>Archetype Created Web Application</display-name>
7    <servlet>
8      <servlet-name>QuoteServlet</servlet-name>
9      <servlet-class>ch.heigvd.amt.mvcsimple.presentation.QuoteServlet</servlet-
           class>
10   </servlet>
11   <servlet-mapping>
12     <servlet-name>QuoteServlet</servlet-name>
13     <url-pattern>/quotes</url-pattern>
14   </servlet-mapping>
15 </web-app>
```

Listing 3.6: The deployment descriptor

Here are some comments about the `web.xml` deployment descriptor:

- line 1-5: this tag specifies the version of the Servlet API used by the application. Warning: be careful when using old examples found on the Web, because specifying an old version here will mean that some mechanisms introduced later will not work (for instance, the expression language used in JSP templates).
- lines 7-10: we define a servlet, by giving it a name and a fully qualified class name.
- lines 11-14: we define a mapping between a URL and a servlet. This is a way to tell the application server: whenever an HTTP request comes in and matches the pattern, then invoke this servlet.
- note that the deployment descriptor is now optional and can be replaced by annotations in the code. You will still encounter this file in older projects (and some teams prefer to declare all attributes in a central location, instead of having the scattered in different source files).

3.3 THE INVERSION OF CONTROL DESIGN PATTERN

The IoC principle is pervasive in client-side and server-side frameworks. In the context of Java EE, it is applied in several tiers. We will see concrete code examples later, but let us start with a presentation of the concept.

3.3.1 *Description of the pattern*

What does it mean to *inverse* the *control*? Control refers to the control of the flow in a program. In simple programs, the developer controls the flow. He writes functions, which may delegate work to other functions, in a top-down fashion. A *library* is a collection of functions that can be shared and reused across programs. The developer using a library decides when to call the provided functions. For instance, when implementing a password management function, the developer might call a function provided by an encryption library.

When using a library, the developer controls the flow. He decides when to call functions provided by the library.

Inverting the flow of control means that the developer does not call a function, but rather provides a function that *will called when it makes sense*. This approach is at the core of object-oriented *frameworks*. A framework implements a generic behaviour with a set of abstract classes that collaborate with each other. The developer creates an application by extending the framework classes and implementing concrete methods. But the developer does not control the flow, the framework does. At some point, the framework decides that it is time to create an instance of the concrete class provided by the developer, or to invoke one of its method.

When using a framework, the developer does not control the flow anymore. He provides code that is called by the framework, when needed.

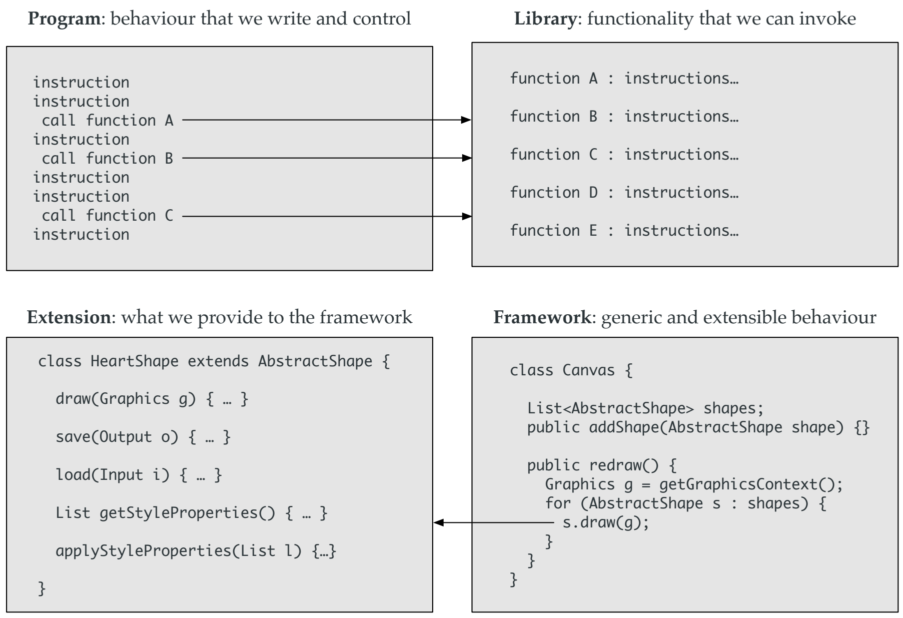
To illustrate this mechanism, think about a graphical editor, allowing the user to create shapes on a canvas and to apply various styles. The application provides a user interface, with menus, windows, palettes, etc. It also implements data management, printing, etc. All these behaviours are implemented by classes, which collaborate with each other. If the graphical editor is implemented as an object-oriented framework, then the developer might be able to augment its behaviour by extending a `Shape` class. In this class, he might implement methods like `draw`, `load`, `save`, `getStyleProperties`, `applyStyleProperties`, etc. The developer provides implementations and understands that it is the framework that will invoke them when needed.

The difference between a library and a framework is shown in Figure 3.2. The pseudo-code indicates that a program *calls* a library, but *is called* by a framework. This is sometimes described as the *Hollywood Principle*.

3.3.2 *IoC in the Java EE presentation tier*

As a matter of fact, we have already seen how the IoC pattern is applied in the presentation tier, when using servlets.

Let us consider how a developer could build a dynamic web application in Java, but without a Java EE application server. In this case, the developer would use classes from the `java.net` and `java.io` packages. He would create a server socket and accept connection requests in a loop. For every new client, he would read the incoming HTTP request line by line. He would prepare a response and send it back to the


Program: behaviour that we write and control
Library: functionality that we can invoke
Extension: what we provide to the framework
Framework: generic and extensible behaviour

```
instruction
instruction
 call function A
instruction
 call function B
instruction
instruction
 call function C
instruction
```

```
function A : instructions…
function B : instructions…
function C : instructions…
function D : instructions…
function E : instructions…
```

```
class HeartShape extends AbstractShape {
  draw(Graphics g) { … }
  save(Output o) { … }
  load(Input i) { … }
  List getStyleProperties() { … }
  applyStyleProperties(List l) {…}
}
```

```
class Canvas {

  List<AbstractShape> shapes;
  public addShape(AbstractShape shape) {}

  public redraw() {
    Graphics g = getGraphicsContext();
    for (AbstractShape s : shapes) {
      s.draw(g);
    }
  }
}
```

Figure 3.2: Inversion of Control: libraries vs frameworks

client, via the socket. Clearly, the developer would control the flow and make calls to classes and methods provided by the Java runtime environment.

The code that we have seen in this chapter follows a different logic. We use the application server as a framework. It implements the generic behavior of any web application: it manages the server socket and accepts client requests. Of course, without us, it is an empty shell and is not able to send meaningful responses to the clients. When we have written the servlet class, we have used an extension point provided by the framework. We have implemented a method that can be called by the application server *at the right time*. But when does the application server decides what is *the right time*? This is where the annotation in Listing 3.1 or the servlet mapping in Listing 3.6 comes in. The developer uses these mechanisms to register his extension and to specify that it should be called when an incoming HTTP request has certain attributes. The URL is first used to identify the servlet class, the HTTP method is then used to identify the method within this class.

3.4 THE PIPES AND FILTERS PATTERN

The Pipes and Filters pattern can be applied when the processing of a task can be decomposed in a number of steps, and when the steps can be handled by independent, reusable building blocks. The pattern is not specific to multi-tiered applications and can be applied in a wide range of situations. One example is the usage of the *pipe* operator to combine unix commands in a sequence. Another example is the use

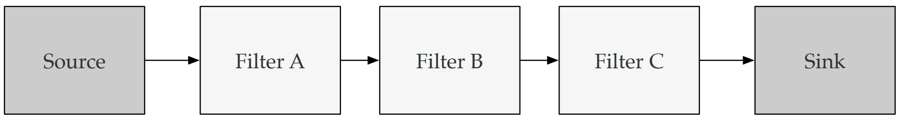


Figure 3.3: The Pipes and Filters design pattern

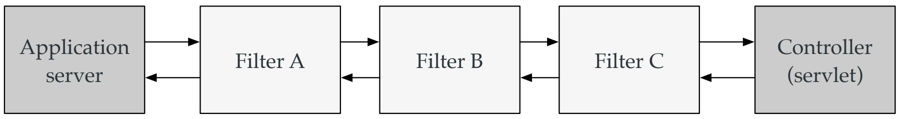


Figure 3.4: The Pipes and Filters design pattern in the presentation tier

of the `Filter` classes in the `java.io` package, which make it possible to apply different transformations when reading and writing data streams.

3.4.1 *Description of the pattern*

The pattern is represented in Figure 3.3. The *source* produces a *stream* of *tasks*, which enter a processing *pipeline*. The *pipeline* is not a monolithic piece of code. Instead, it is created by assembling a series of *filters*. Every *filter* performs a well-defined function on the *task*. This means that the *task* can be modified or augmented before being passed to the next *filter*. At the end, the sink receives the result of the computation. The great thing about this pattern is that it is possible to reuse *filters* in different types of *pipelines*.

3.4.2 *Pipes and Filters in the presentation tier*

When applying the pattern in the presentation tier, the tasks processed by the pipeline are the HTTP requests. The application server, which receives the request, is the source. HTTP requests are processed by several filters before they reach the controller. In fact, as shown in Figure 3.4, the pattern is applied twice: HTTP responses produced by the controller also go through the sequence of filters, which have the opportunity to modify them.

But what is the reason for using filters and what kind of processing is typically done in these components? These are common use cases:

- *auditing*: a filter can just observe the requests and generate an audit log
- *authorization*: a filter can decide if the user making the request has the right to do it
- *compression*: a filter can decompress incoming requests and compress outgoing responses
- *encryption*: a filter can decrypt requests and encrypt responses

- *caching*: a filter can optimize performance by reusing previously generated responses (in this case, the filter may short-circuit the pipeline)

In order to create custom servlet filters, the developer needs to implement the `Filter` interface, which defines three methods: `init(FilterConfig filterConfig)`, `destroy()` and most importantly `doFilter(ServletRequest request, ServletResponse response, FilterChain chain)`. An example is shown in Listing 3.7. Here are some explanations about the code:

- line 7: the annotation registers the filter, so that IoC can happen
- line 8: we implement the `Filter` interface
- line 12: `doFilter` is called when the requests advances in the pipeline. Other filters might have been called before, other filters might be called after. The `chain` parameter represents the processing pipeline.
- line 19: to test the pipeline, we simply attach a String model to the request. This model will be available in other filters, in the controller servlet and in the JSP template.
- line 21: we pass the request further down the chain.
- line 23: when the call returns, we have the ability to inspect the response and to modify it. However, if we want to do that, we need to wrap `resp` in a special object, to capture the output stream in memory.

```java
package ch.heigvd.amt.mvcsimple.presentation;

import javax.servlet.*;
import javax.servlet.annotation.WebFilter;
import java.io.IOException;

@WebFilter(filterName = "CustomFilter", urlPatterns = "/*")
public class CustomFilter implements Filter {
    public void destroy() {
    }

    public void doFilter(ServletRequest req, ServletResponse resp, FilterChain
         chain) throws ServletException, IOException {
        /*
         The pipeline is first going from the app server towards the controller.
         We can log the request, transform the request. We can also block the
         request (by not calling chain.doFilter(req, resp);
         */

        req.setAttribute("computedByFilter", "yellow");

        chain.doFilter(req, resp);

        /*
         We are on the way back. Now, we can transform the response. But to
         do that, we would need to send a wrapper to the chain in the previous
         call. You can find an example in the Java EE 5 tutorial
         */
    }

    public void init(FilterConfig config) throws ServletException {
    }

}
```

Listing 3.7: A servlet filter example

3.5 QUESTIONS

To answer these questions, you will need to have read the chapter but also to have done some research. Make sure that you are able to answer every question. Discuss your responses with your peers.

1. *How does one implement HTML form processing in Java EE?* Consider the scenario where the user arrives on a page, fills out three fields (`first name`, `last name` and `e-mail address`) and presses a `Register` button. We want to process the event on the server side and to validate the data entered by the user (the fields cannot be empty and the e-mail address must contain an @ sign). What code do you have to write?

2. *What is the difference between the request, session and application scopes in the servlet API?* The method `setAttribute` is available in the `HttpServletRequest`, `HttpSession` and `ServletContext` classes. What is the difference and when should they be used?

3. *Why do we have to be careful with the `HttpSession` object?*. The Servlet API makes it very easy to store objects in the session. While it is practical to use this feature to build stateful applications on top of the stateless HTTP protocol, it does not come for free. There are at least risks or drawbacks to be aware of. What are they?

4. *How do we deal with relative links in JSP templates?*. It is possible to deploy several applications in the same applications server. For this to work, every application is assigned a *context root*. In order to generate links to CSS files and other assets, it is often necessary to include the context root in the path and it is a bad idea to hard-code the value in the template. What is the solution to this problem?

4 THE BUSINESS TIER

4.1 INTRODUCTION

In the previous chapter, we have looked at what happens in the client tier (on the user device) and in the presentation tier (on the server side). We have seen the MVC pattern in details and seen how it can be applied to generate dynamic HTML pages on the server side. We have seen that this pattern does not involves three components, but four: in addition to the *model*, the *view* and the *controller*, there is also a *service*. The role of the service is gather data and to apply business logic to generate the model, which is finally rendered by the view.

While it is possible to implement business services with POJOs, we introduce the notion of managed component. It allows the container to apply various policies (pooling, security, transactions, etc.).

This chapter focuses on this type of component and describes how it can be implemented with Java EE technologies. As always, there are different solutions and every solution has benefits and drawbacks. In the second part of the book, we will see how business services can be implemented with Plain Old Java Object (POJO)s managed by an application framework like Spring. In this part of the book, we present the approach recommended by the Java EE specification, namely the use of EJBs.

4.2 ENTERPRISE JAVA BEANS

The use of EJBs has been a heated topic in the developer community and many discussions can be found on that subject in online forums. The accuracy of the arguments in these debates vary a lot and need to be taken with a grain of salt.

When you implement an application with Java EE, you do not have to use EJBs. When you build an application with the Spring framework, you do use some of the Java EE technologies.

For instance, one often hear statements like "I don't want to use EJBs, therefore I don't want to use Java EE, therefore I will use something more lightweight like the Spring Framework". *This statement is incorrect!* The EJB specification is one of the many specifications that are part of the Java EE umbrella specification. It sits at the same level as the Servlet API or the JSP specification. Deciding to use Java EE to build an application does not mean that every single sub-specification *has to* be used. Depending on the functional and non-functional requirements, the developer has to select the components that make sense in a given context. Coming back to the statement, as we will see in the second part of the book, the Spring Framework is built on top of technologies that are in the Java EE scope. Spring applications can be packaged in .war files and deployed in application servers. *Hence, when building a web application with the Spring framework, one does use some of the Java EE APIs.*

4.2.1 *Opinions on EJBs: perceptions vs facts*

Be careful when reading articles about a platform that has evolved over 20 years. What was true in the early days might be very different today.

When seeking for opinions on EJBs in online forums and blog posts, one also need to be careful about the date when the comments were made. Both the *specification* and the *implementations* of EJBs have changed dramatically since the beginning and we are talking about a technology that has matured over 15 years:

- The evolution of the *specification* has a big positive impact on the developer experience. Developing EJBs with the first versions of the specification was a tedious projects. Several Java and XML files had to be created for every single business component. With the current specification, creating an EJB can be done by adding a single annotation in a single Java file. The version 3.0 of the specification, which was part of Java EE 5, was a major shift which happened in 2006.
- The evolution of the implementations has a positive impact on systemic qualities and particularly on performance. As we will see, EJBs handle a lot tasks behind the scenes and it is not surprising that calling an EJB is more expensive than calling a POJO. Twenty years ago, the performance overhead was much more perceptible than it is today.

Online archives can give the perception that EJBs are still a pain to use and slow, but often people just repeat what they have heard and read. Only experiments with up-to-date environments can give the real picture. EJB may not be the best choice for a given project, but the decision to use or not to use them should be based on facts rather than impressions. This comment applies to any architectural decision.

4.2.2 *The original vision*

The authors of the EJB specification wanted to define a standard way to create and operate business services. Business-focused developers would create independent components, such as a customer management service, an inventory management service or an order management service. When creating these components, the developers would focus purely on the business logic and not deal with any user interface concern. The set of business components could then be used in two ways:

- they could be combined to create a complete *enterprise application package*. Concretely, every business service would be packaged in an *EJB module* (a `.jar` file). Several of these modules would be added to an *enterprise archive* (an `.ear` file). A `.war` file, packaging presentation-tier components would also be added to the enterprise archive. In this scenario, the development team would provide a big `.ear` file to the IT operations team, which would deploy it into an application server (or rather into a cluster of application servers).

- they could also be deployed individually into the application server. A fundamental aspect of the original vision was that EJBs would be *distributed* components, exposing a remote interface. Hence, it should be possible for an application component in any tier to locate and invoke these business services. For instance, a rich client application should be able to make remote calls to an EJB without going through servlets. Also, one EJB running on a machine should be able to make remote calls to another EJB running elsewhere.

The two scenarios are shown in Figures 4.1 and 4.2. In the first case, a complete enterprise package has been created and it is deployed across the cluster as a unit. Note that every EJB may still expose a remote interface and be accessed from any application server node and from client machines. But in the normal flow, HTTP requests are processed by components in the `.war` package, which generates calls to EJBs in the `.jar` files. In the second case, the `.war` and individual `.jar` files are deployed individually. Benefits of this approach is that services can be updated and redeployed with greater autonomy, and that flexible scaling strategies are possible. However, they come at the cost of increased complexity. The bottom line is that you make the choice to deploy an EJB module as an independent unit, there must be a clear reason to do it.

In general, developers package all components together and deliver an application package. Since Java EE 6, it is not mandatory to use the .ear format: EJBs can be packaged in a .war file.

4.2.3 *Types of Enterprise Java Beans*

So far, we have used the term EJB in a generic way, to refer to components that implement business services and that can be deployed in application servers. In fact, the specification defines several types of beans: StateLess Session Bean (SLSB), StateFul Session Bean (SFSB), Message Driven Bean (MDB) and now defunct Entity Beans (not to be confused with entities in the JPA specification.

Developers tend to use stateless session beans a lot more than stateful session beans. Message driven beans are very useful when the application has to be integrated with legacy applications. Entity beans have disappeared and been replaced with entities defined in the Java Persistence API.

4.2.3.1 *Stateless Session Beans*

These are the most commonly used EJBs are used to implement services that do not maintain a conversational state. In other words, when every method call made by a client on a SLSB is independent of the previous calls.

When creating a EJB, one has to decide whether it will be accessed *locally* (from servlets and other EJBs in the same application), or *remotely* (from rich clients and other applications). For many applications, the

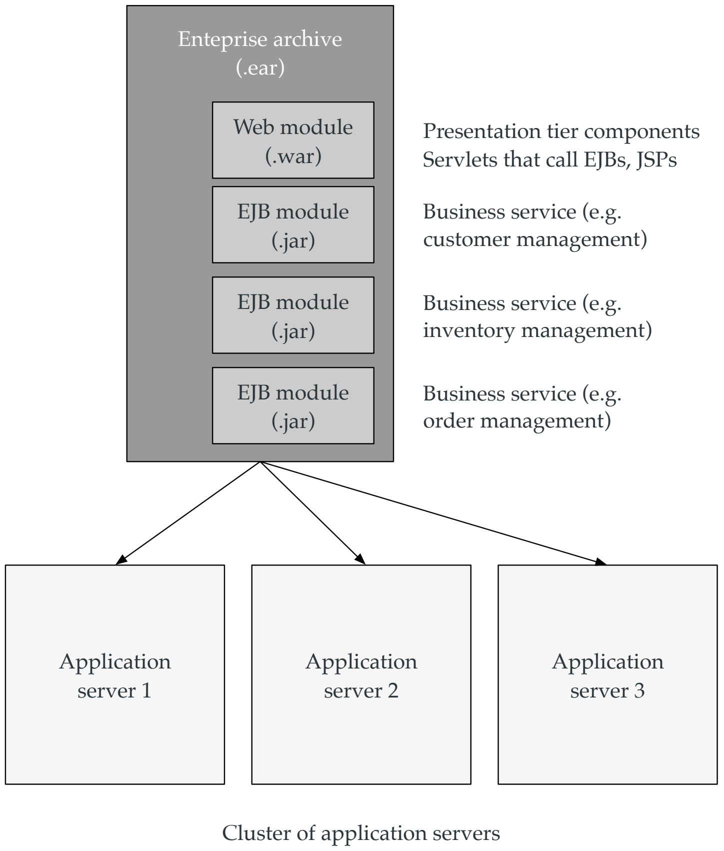


Figure 4.1: Deployment of a complete enterprise application in a cluster

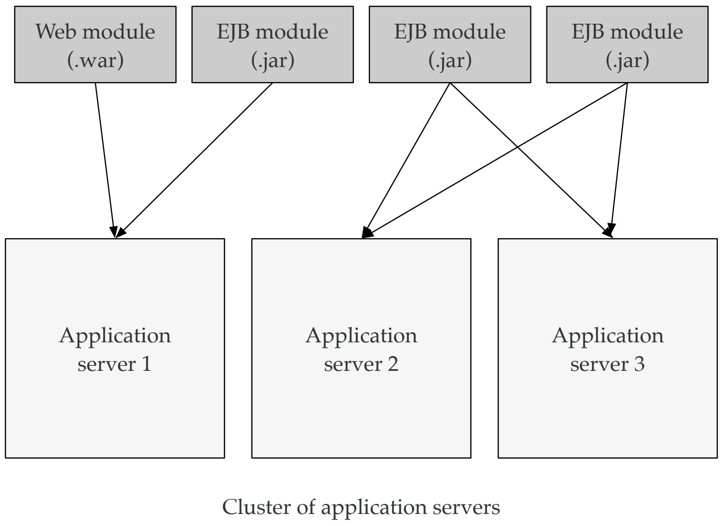


Figure 4.2: Deployment of individual components in a cluster

default choice is now to expose only a local interface. When the decision has been made, the interface can be created and annotated either with `@Local` and `@Remote`. The next step is to implement a Java class, which does not have to extend any class but has to be annotated with `@Stateless`. This is what will allow the application server to detect that the class is a managed component, which it must instantiate and control.

Since Java EE 6, it is not mandatory to explicitly local interfaces for SLSBs and it is possible to only create a class. Still, we consider that it is a good practice to create interface files.

```java
package ch.heigvd.amt.mvcsimple.business;

import javax.ejb.Local;

@Local
public interface ICustomerManager {

    public String generateCustomerId();

}
```

Listing 4.1: Local interface defined for a stateless session bean

```java
package ch.heigvd.amt.mvcsimple.business;

import javax.ejb.Stateless;
import java.util.UUID;

@Stateless
public class CustomerManager implements ICustomerManager{

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

}
```

Listing 4.2: Stateless session bean implementation

4.2.3.2 *Stateful Session Beans*

SFSBs are beans that manage conversational state. This means that some state is shared between the successive method call made by one client on the bean. For instance, it is possible to implement a shopping cart service, exposing methods like `void addProductToCart(Product p)` and `List<Product> getCartContent()`. Calling the first method 3 times before calling the second method shows that a state has to be managed during the conversation. Instead of storing it in the presentation tier or in the database, it is placed under the control of the SFSB. This raises interesting questions in terms of resource consumption and scalability, which the specification addresses with mechanisms that we will not describe in details (e.g. *passivation* and *activation*).

Why manage conversational state in a SFSB rather than in an HTTP session? One argument is the need to support rich client that do not go through the presentation tier. Another argument is the separation of UI navigation state machines from business workflow state machines.

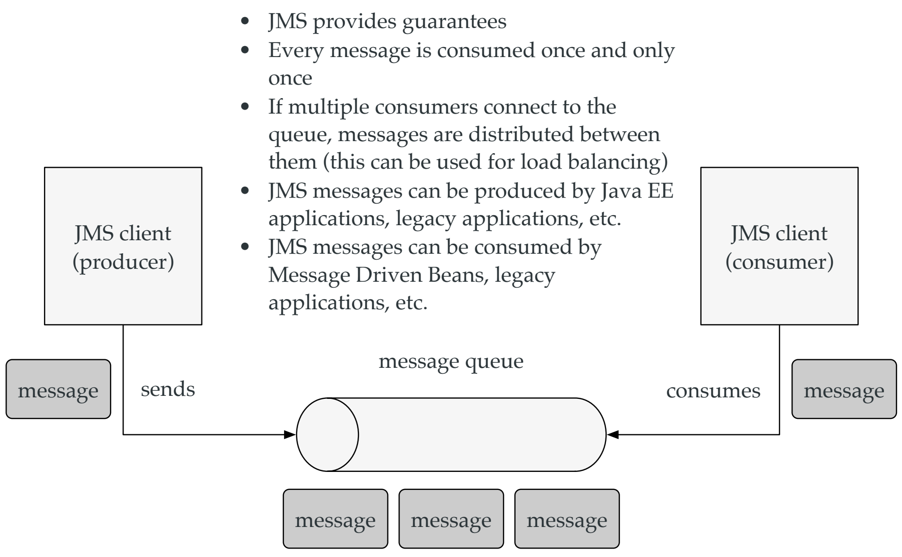

JMS provides guarantees
Every message is consumed once and only once
If multiple consumers connect to the queue, messages are distributed between them (this can be used for load balancing)
JMS messages can be produced by Java EE applications, legacy applications, etc.
JMS messages can be consumed by Message Driven Beans, legacy applications, etc.
JMS client (producer)
JMS client (consumer)
message
sends
message queue
consumes
message
message
message
message

Figure 4.3: Point-to-point communication with a JMS queue

SFSBs are used much less often than SLSBs. Even if there is often a need to manage conversational state, developers prefer to manage it in a different tier. They tend to either manage it in the presentation tier, with the HTTP session abstraction, or in the integration tier, with distributed data stores such as Redis. A couple of years ago, the JBoss Seam framework was perhaps the strongest advocate of SFSBs and its documentation provides valuable information on the topic.

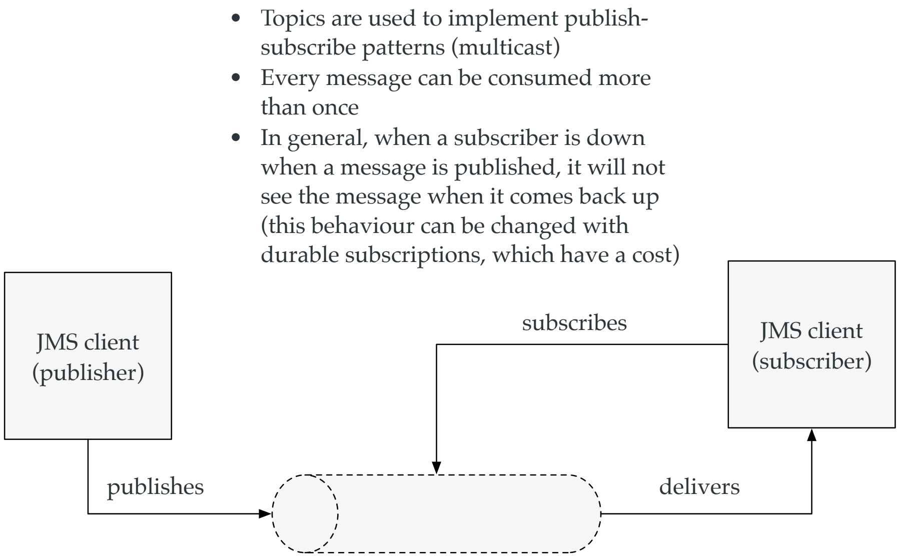
4.2.3.3 *Message Driven Beans*

MDBs are a great solution for integration with legacy applications and asynchronous task processing. In recent years, competing approaches have appeared in the NoSQL and big data space (redis, kafka, pulsar, etc.)

MDBs are very useful beans that are related to the JMS specification, which deals with asynchronous messaging across components and applications. JMS provides abstractions for point-to-point messaging (with queues) and publish-subscribe (with topics). A MDB is a service that does not react to an incoming HTTP request or to a (remote) procedure call. It is a service that reacts to a message posted in a queue or broadcasted on a communication channel. It implements a single method, with the following signature: `public void onMessage(Message message)`. Messages exchanged via JMS have a payload and metadata.

One common scenario to use MDBs is the integration of a Java EE application with other enterprise applications. For instance, think of a new application that needs to interact with a legacy application in a banking information system. Using a messaging system, with point-to-point queues or publish-subscribe channels is an approach with many benefits: low coupling, asynchronous interactions, etc.

Another common pattern is to implement a form of load balancing at the application level. Think of an application, where users can submit tasks (e.g. the task to look for a cheap flight, the task to



Topics are used to implement publish-subscribe patterns (multicast)
Every message can be consumed more than once
In general, when a subscriber is down when a message is published, it will not see the message when it comes back up (this behaviour can be changed with durable subscriptions, which have a cost)
JMS client (publisher)
subscribes
JMS client (subscriber)
publishes
delivers

Figure 4.4: Publish-Subscribe communication with a JMS topic

generate a report, etc.). Instead of making a synchronous call to a service and to wait for an answer, it is often better to capture the task in an object and to process it asynchronously. A message queue can be used to notify worker nodes that tasks have been submitted. The semantics of JMS queues guarantee that every message posted to a queue will be consumed once and only once. If multiple workers connect to the same queue and consume messages, the system will deliver part of the messages to each worker. Hence, it is possible to scale an application by adding more worker nodes behind a queue. This is one reason for packaging and deploying MDBs independently from a full enterprise archive (in other words, to package them in a separate `.jar` file and not within the `.ear` file).

4.2.3.4 *Entity Beans*

Entity beans were EJBs. They are not the same thing as entities defined in the Java Persistence API, which we will see later.

Entity beans are a thing of the past. In the original specification, they were proposed as a way to interact with the database. They were a major pain to work with. This was true when designing the application, because they made it very hard to build an object-oriented domain model. This was also true when running the the application, because they raised a lot of performance issues.

The rapid success of the Hibernate ORM, followed by the standard Java Persistence API, became the preferred way to interact with the database in Java EE. Entity beans can only be found in very old applications.

4.3 MANAGED COMPONENTS

To understand how EJBs work and the value they add to POJOs, it is first necessary to understand the concept of *managed component*. This concept is related to the IoC pattern that we have seen in the previous chapter. Remember that when we have introduced it, we have explained how application frameworks work and how they control the creation of classes and the invocation of methods provided by the developer.

4.3.1 *Lifecycle management*

The developer does not create instances of managed components: the framework does.

Managed components have a well-defined *lifecycle*, which can be described by a state machine. The *state machine* specifies all the states in which the component can be, from its initial creation until its final destruction. The state machine also specifies what are the *events* that cause state transitions and what is the origin of these events. We say that components are managed because the application developer is not responsible for controlling their lifecycle. The framework, in our case, the Java EE container, is responsible for controlling it.

When we have studied the presentation tier, we have seen a first type of managed component: the servlets. Remember that in the code that we wrote, we implemented sub-classes of `HttpServlet`, but we never created instances of these classes. The application server took care of this task. The lifecycle of servlets is precisely described in the Servlet specification and this is how we can truly understand when servlet instances are created, how many instances are created and how they behave from a concurrency point of view. Understanding this lifecycle is mandatory to write correct and high-performance code. In the third part of the book, Section 6.3 describes an experiment that shows how misunderstandings can lead to bugs that manifest themselves under load.

4.3.2 *Resource pooling*

With a pool, it is possible to reuse objects that are expensive to create. It is also possible to limit the number of instances that can be created and to keep control of resources.

Depending on the context, resource pooling can mean quite different things. Here, we are considering scenarios where special objects can be shared and reused across multiple invocations. Examples of objects that are often pooled include threads, database connections, network connections. There are different reasons why managing a pool of objects is interesting:

- It *improves performance* when the creation of objects is expensive. A good example is a database connection. Creating one takes a lot of time, because it is necessary to establish a connection, to send credentials and to initialize the conversation. Instead of doing all that work for every client request, the idea is to keep several connections in a pool. When a new client request arrives, we can get a connection from the pool, use it to process the request and return it to the pool (without closing it).

- It *prevents resource exhaustion* and allows us to limit the rate at which client requests are served. In the example of database connections, if a new connection is opened for every client request, there is a risk to overload the database if many client requests arrive at the same time. A pool can be used to limit the concurrency: when a client request comes in and there is no connection available in the pool, then the request is placed in a queue until one becomes available.

The application server is managing EJBs in pools. The behaviour of the EJB pools can be configured in the administration interface of the application server. Typically, the initial size of the pool, its maximum size, and what happens when the pool has been exhausted can be configured. This is an area where there are differences between application servers.

4.3.3 *Aspect oriented programming*

With AOP, it is possible to split the source code in multiple files. Functional aspects are implemented in one file, non-functional aspects are implemented in other files. This increases modularity, facilitates reuse and maintainability.

One design goal for the Java EE and the EJB specifications was to facilitate *separation of concerns*. Separation of concerns means that in a well-designed codebase, every unit should focus on a single task (a single concern, a single aspect). Aspect Oriented Programming (AOP) is a programming paradigm that supports this goal. EJBs implement a basic form of AOP. Other frameworks, such as AspectJ and the Spring framework, offer more advanced mechanisms.

Enterprise applications are complex for two reasons. Firstly, they are complex because they implement rich domain models, with a lot of business entities and processes. Secondly, they are complex because they have to fulfill non-functional requirements: security, availability, performance, etc. The skills needed to address these two types of complexity are very different. To address the first one, developers need deep business domain expertise. To address the second one, developers need deep technical expertise.

Is there a way for a platform to take care of the technical complexity, thereby letting the developers focus only on the business complexity? This was the vision for the Java EE platform: managed components were seen as a way to let the application server deal with the technical aspects, behind the scenes. The developers would merely have to *declare their intents*, originally in deployment descriptors and later on in annotations. In reality, it is not possible to design a multi-tiered application without understanding distributed computing issues and tradeoffs. Application developers have to deal with *some* of the techni-

```
private Context ctxt;

public void transferMoney(Account a, Account b, double amount) {

  // authentication concern
  if (!ctxt.isAuthenticated()) {
    thrown new UnauthorizedException();
  }

  // authorization concern
  Principal p = ctxt.getPrincial();
  if (!a.getOwner().equals(p)) {
    throw new UnauthorizedException();
  }

  // audit concern
  AuditLog.getLog().record('transfer', p, a, b);

  // tx concern
  ctxt.getTxManager().begin();

  // business logic concern
  try {
    a.debit(a, amount);
    a.credit(b, amount);
  } catch (Exception e) {
    // tx concern
    ctxt.getTxManager().rollback();
  }

  // tx concern
  ctxt.getTxManager().commit();

}
```

Figure 4.5: Code without separation of concerns

```
private Context ctxt;

@RolesAllowed("customer")
@TransactionAttribute(REQUIRED);
@Interceptors(AuditInterceptor.class);
public void transferMoney(Account a, Account b, double amount) {

  // authorization concern
  Principal p = ctxt.getPrincial();
  if (!a.getOwner().equals(p)) {
    throw new UnauthorizedException();
  }

  // business logic concern
    a.debit(a, amount);
    a.credit(b, amount);

}
```

Figure 4.6: Code with separation of concerns

cal complexity. But it is true that managed components, such as EJBs, can *reduce* the technical complexity and handle some of work.

To illustrate the notion of separation of concerns, let us compare the code in Figure 4.5 and Figure 4.6. In the first case, we see that the business logic is lost in the middle of a lot of statements that deal with authentication, authorization, auditing and transaction management. The code is not easy to read and it is hard to grasp the business intent of the method at first glance. In the second case, the non-functional statements have been removed and replaced by annotations. In other words, the developer has used a way to declare how the method should behave from a security and transactional point of view. This is information that the developer gives to the application server, who will have the responsibility to enforce these policies. In the example, we see that some aspects can be declared with standard EJB annotations (`@RolesAllowed` and `@TransactionAttribute`), while other aspects need to be handled by the developer but in a separate class: the `@Interceptor` annotation states that the class `AuditInterceptor`, provided by the developer, has to be applied as a filter when the method is invoked. This is an application of the pipes and filters pattern that we have seen in Section **??**.

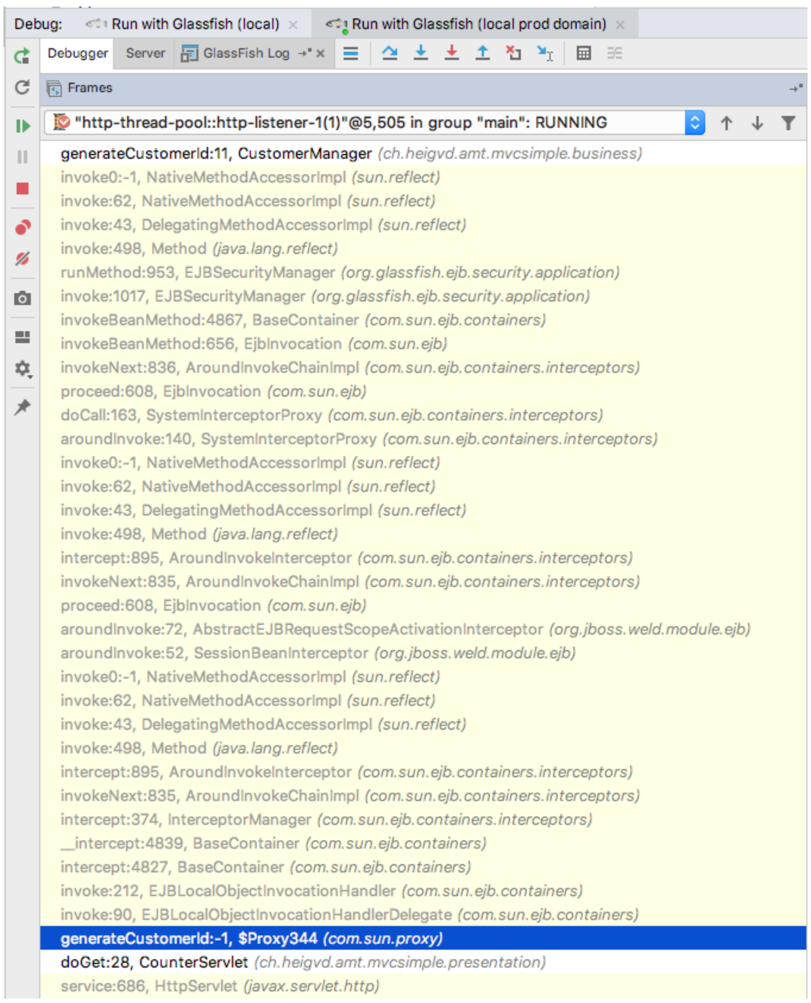
4.3.4 *Aspect oriented programming: behind the scenes*

Using AOP mechanisms when implementing EJBs is fairly easy. One needs to know what aspects are supported by the platform and know which annotations are used to apply them. But it is also important to understand what happens behind the scenes. How is it possible for the container to apply these policies when a client invokes a method on an EJB?

The following experiment can reveal some of the work done by the application server. Run the application with the debugger enabled and simply put a breakpoint in the SLSB method. Access the servlet that makes a call to the SLSB and have a close look at the stack trace in your IDE. An example is shown in Figure 4.7.

Never, ever, create an instance of an EJB yourself. You would bypass the creation of a dynamic proxy and the magic would disappear.

In this example, we see that two methods named `generateCustomerId()` have been called. As could be expected, one them is the method that we implemented in our `CustomerManager.java` class. The other, however, is not code that we wrote. Instead, we see that it is provided by a class named `$Proxy344` in the `com.sun.proxy package`. The container uses the mechanism of dynamic proxies offered by the Java platform. It can intercept every method call made to the bean and can inject additional logic.


Debug:
Run with Glassfish (local)
Run with Glassfish (local prod domain)
Debugger
Server
GlassFish Log
Frames

```
"http-thread-pool::http-listener-1(1)"@5,505 in group "main": RUNNING
generateCustomerId:11, CustomerManager (ch.heigvd.amt.mvcsimple.business)
invoke0:-1, NativeMethodAccessorImpl (sun.reflect)
invoke:62, NativeMethodAccessorImpl (sun.reflect)
invoke:43, DelegatingMethodAccessorImpl (sun.reflect)
invoke:498, Method (java.lang.reflect)
runMethod:953, EJBSecurityManager (org.glassfish.ejb.security.application)
invoke:1017, EJBSecurityManager (org.glassfish.ejb.security.application)
invokeBeanMethod:4867, BaseContainer (com.sun.ejb.containers)
invokeBeanMethod:656, EjbInvocation (com.sun.ejb)
invokeNext:836, AroundInvokeChainImpl (com.sun.ejb.containers.interceptors)
proceed:608, EjbInvocation (com.sun.ejb)
doCall:163, SystemInterceptorProxy (com.sun.ejb.containers.interceptors)
aroundInvoke:140, SystemInterceptorProxy (com.sun.ejb.containers.interceptors)
invoke0:-1, NativeMethodAccessorImpl (sun.reflect)
invoke:62, NativeMethodAccessorImpl (sun.reflect)
invoke:43, DelegatingMethodAccessorImpl (sun.reflect)
invoke:498, Method (java.lang.reflect)
intercept:895, AroundInvokeInterceptor (com.sun.ejb.containers.interceptors)
invokeNext:835, AroundInvokeChainImpl (com.sun.ejb.containers.interceptors)
proceed:608, EjbInvocation (com.sun.ejb)
aroundInvoke:72, AbstractEJBRequestScopeActivationInterceptor (org.jboss.weld.module.ejb)
aroundInvoke:52, SessionBeanInterceptor (org.jboss.weld.module.ejb)
invoke0:-1, NativeMethodAccessorImpl (sun.reflect)
invoke:62, NativeMethodAccessorImpl (sun.reflect)
invoke:43, DelegatingMethodAccessorImpl (sun.reflect)
invoke:498, Method (java.lang.reflect)
intercept:895, AroundInvokeInterceptor (com.sun.ejb.containers.interceptors)
invokeNext:835, AroundInvokeChainImpl (com.sun.ejb.containers.interceptors)
intercept:374, InterceptorManager (com.sun.ejb.containers.interceptors)
__intercept:4839, BaseContainer (com.sun.ejb.containers)
intercept:4827, BaseContainer (com.sun.ejb.containers)
invoke:212, EJBLocalObjectInvocationHandler (com.sun.ejb.containers)
invoke:90, EJBLocalObjectInvocationHandlerDelegate (com.sun.ejb.containers)
generateCustomerId:-1, $Proxy344 (com.sun.proxy)
doGet:28, CounterServlet (ch.heigvd.amt.mvcsimple.presentation)
service:686, HttpServlet (javax.servlet.http)
```

Figure 4.7: The debugger reveals thE proxy created by the container

4.4 THE DEPENDENCY INJECTION PATTERN

The last topic that we cover in this chapter is actually a generic design pattern, which can be applied in every tier of the architecture. For instance, on the client-side, the Javascript framework AngularJS makes extensive use of the pattern.

When object A calls a method on object B, we say that A depends on B. How does A obtain the reference to B?

Before describing the pattern, it is necessary to clarify what we mean by *dependency*. Our context is object-oriented programming. An object has a dependency on another object if at some point it needs to invoke one of this methods. For instance, a `CheckoutService` class might have a dependency on a `PriceCalculatorService` class: in an online shop, when you proceed to checkout, the price of your order needs to be computed (which can be a fairly sophisticated process with various policies and business rules). Very often, dependencies are not expressed on concrete classes, but rather on interfaces. In the previous example, this would make it possible to implement different pricing strategies in different classes and to select one of them at runtime.

Now that we know what a dependency is, we can explain what it means to *inject* one.

- An example of code that does not use dependency injection is shown in Listing 4.3. In this case, the `CheckoutService` obtains its dependency by instantiating it itself. This creates a tight coupling between the two classes and does not make it possible to switch the implementation at runtime.

- A better approach is shown in Listing 4.4. In this case, the client uses a service registry and performs a lookup operation. It asks the registry to obtain a reference to the dependency by providing a service name. This technique was the standard way to manage dependencies between servlets and EJBs in the first versions of the platform.

- Finally, dependency injection is shown in List 4.5. In this case, the developer only declares that the checkout service has a dependency on the pricing service. He uses an annotation provided by the dependency injection framework (for a Java EE container, he uses the `@EJB` annotation). This works if and only if the services are managed components, whose lifecycle is controlled by the container. At runtime, the container creates instances of `CheckoutService` and of an implementation of`IPricingService`. It scans the annotations and sees that it must inject a reference to the second instance in the first instance. How this is done depends on the technology and framework, but common techniques are *constructor-based dependency injection* (the constructor of the dependent class includes a parameter so that a reference to the dependency can be passed) and *setter-based dependency injection* (a setter method is provided by the developer and invoked by the framework).

Once again: never, ever, create an instance of an EJB yourself. Either use dependency injection or do a lookup in the naming registry.

Last but not least, Listing 4.6 shows a concrete application of dependency injection in the Java EE platform. The container is managing two components: a servlet and a SLSB. The developer has expressed the fact that the servlet depends on the SLSB by using the `@EJB` annotation in the servlet. This is what allows the container to figure out what needs to be done and how it should wire the components together.

```java
package ch.heigvd.amt.mvcsimple.business;

public class CheckoutService {

    private IPricingService pricingService;

    public CheckoutService() {
        pricingService = new PricingServiceImpl1();
    }

    public void checkout(Cart cart) {
        pricingService.computePrice();
    }

}
```

Listing 4.3: Bad practice: dependency is instantiated by the client service

```java
package ch.heigvd.amt.mvcsimple.business;

public class CheckoutService {

    private IPricingService pricingService;

    public CheckoutService() {
        pricingService = (IPricingService) ServiceRegistry.lookup("pricing-service
             -policy-xy");
    }

    public void checkout(Cart cart) {
        pricingService.computePrice();
    }

}
```

Listing 4.4: Better practice: dependency is looked up in a registry

```java
package ch.heigvd.amt.mvcsimple.business;

public class CheckoutService {

    @AnnotationProvidedByTheDependencyInjectionFramework
    private IPricingService pricingService;

    public CheckoutService() {
    }

    public void checkout(Cart cart) {
        pricingService.computePrice();
    }

}
```

Listing 4.5: Dependency is injected by the framework as part of the lifecycle

```java
package ch.heigvd.amt.mvcsimple.presentation;

import ch.heigvd.amt.mvcsimple.business.ICustomerManager;

import javax.ejb.EJB;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;

@WebServlet(name = "UserManagerServlet", urlPatterns = "/users")
public class UserManagerServlet extends HttpServlet {

    @EJB
    ICustomerManager customerManager;

    protected void doGet(HttpServletRequest request, HttpServletResponse response)
          throws ServletException, IOException {
        String newCustomerId = customerManager.generateCustomerId();
        request.setAttribute("newCustomerId", newCustomerId);
        request.getRequestDispatcher("/some/view").forward(request, response);

    }
}
```

Listing 4.6: Injecting a reference to a SLSB into a servlet

4.5 QUESTIONS

To answer these questions, you will need to have read the chapter but also to have done some research. Make sure that you are able to answer every question. Discuss your responses with your peers.

5 THE INTEGRATION TIER

5.1 INTRODUCTION

With what we have seen so far, we are able to write server-side components that handle requests from client programs. We know how to structure our code in the presentation tier with the MVC design pattern. We also know that we can used *managed components*, for instance EJBs to implement the business logic of our application. However, in the examples that we have seen so far, all the data was either generated on the fly or kept in memory data structures. Of course, for real applications we need to be able to interact with persistent data stores, should they be relational or NoSQL databases. The technologies that provide a link between the business services and the databases are located in the *integration tier*, which is the focus of this chapter.

With JDBC, the developer writes SQL queries and sends them to the database. He iterates over tabular results and writes logic to convert records into objects.

We first introduce the Data Access Object (DAO) design pattern, which is used to decouple the business logic from specific database management systems. We also explain how this pattern can be implemented with a combination of Java EE technologies: SLSB to implement the data access services and Java DataBase Connectivity (JDBC) to submit SQL queries to a database. We also show how the *dependency injection* is used once more, this time to inject a reference to a *data source* into the data access service.

With an ORM, the developer declares how his object-oriented model is mapped to a set of tables. The ORM generates SQL queries and handles the conversion between records and objects.

In the second part of the chapter, we introduce the notion of ORM, which aims at simplifying the way object-oriented programs interact with relational databases. We illustrate it with the JPA standard, which is part of Java EE. We describe how the API offers and alternative to JDBC. We review some of the annotations defined in the specification and discuss some of the tradeoffs to be aware of when deciding to use or not to use an ORM solution, and when deciding which features of this solution to use.

5.2 ACCESSING DATABASES FROM BUSINESS SERVICES

Let us consider the example of an online shop. One of the use cases is for the customer to land on a *Products* page see a list of products: a small image, a title, a price, ratings, etc. To implement the use case with Java EE, we could go through the following steps:

- write a servlet that handles the HTTP requests sent to `/products`
- write a `Product` POJO class with getter methods

- write a `ProductsManager` SLSB with a `List<Product> getProducts()` method
- write a JSP that receives a list of products and renders it in HTML markup
- wire everything in the servlet, which should obtain a reference to the SLSB (usually via dependency injection), make a call to `getProducts()`, attach the result as model on the request, forwarded to the JSP.

There is one remaining question however: how should we implement the `getProducts()` method. This is where we will need to interact with the database. Before looking at solutions, let us consider what are important aspects to consider:

- *We want to keep our options open and avoid vendor lock-in*. If we write a large application and decide to use a certain type of database, we would like to be able to replace that database with another one in the future. This might mean replacing a Relational DataBase Management System (RDBMS) with another RDBMS (e.g. replace IBM DB2 with PostgreSQL, or replacing MySQL with Oracle Database). This might also mean replacing a RDBMS with a NoSQL database, or vice-versa.
- *We want to increase the productivity of developers*. Depending on the solution, the developers will have to write more or less code. A solution that automates some of the work and reduces the need for boilerplate code will have a positive impact on productivity.
- *We want to have control on non-functional aspects*. If we use a solution that handles some of the work for us, then we must be aware of its impact on performance, scalability and security. We must understand how we can use the right options to control these aspects.

5.2.1 *The Data Access Object pattern*

The DAO pattern is extensively used in multi-tiered applications. Whatever the technology stack, you will often encounter codebases with classes following naming conventions based on a variation of `ProductsDAO`, `ProductsManager`, `ProductsRepository` or `ProductsService`.

These classes will implement an interface exposing `CRUD` operations, such as:

- a method to *create* and store an object in the database
- a method to *update* an object in the database
- a method to *delete* an object from the database

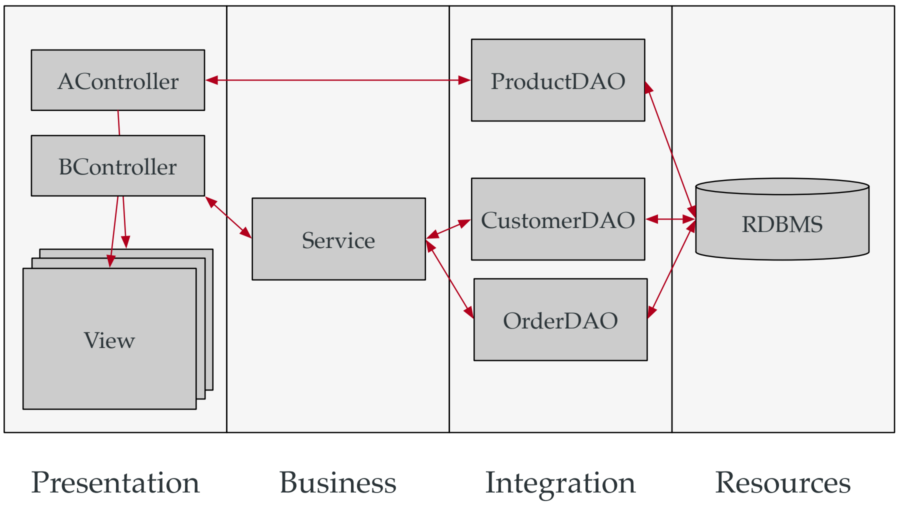


Figure 5.1: The DAO design pattern

- several methods to *read* one or more objects from the database: by providing a unique id, by providing filtering criteria, etc.
- possibly additional methods to get the number of objects in the database, perform batch operations, etc.

Applying the DAO pattern allows us to keep the code specific to a particular database technology in a set of well-defined classes. These classes deal only with this *aspect* of the application and are a good example of *separation of concerns*. The way to connect to the database, the way to send queries to the database, the way to process the results are not scattered across the whole codebase. The business services are isolated from the database layer, because they use methods defined in the DAO interface. In theory, replacing a database with another database can be as simple as implementing a new DAO and injecting a reference to it in the business service. In practice, replacing the database technology may impact non-functional aspects and require some additional work (e.g. to change the frequency at which the DAO methods are called).

5.2.2 *Do we always need a service in the business tier?*

As suggested in Figure 5.1, some use cases may not require a the implementation of a separate service in the business tier. Indeed, it is possible for a controller in the presentation tier to directly access a DAO in the integration tier. But then, how does one decide when to use or to bypass the business tier? There is no clearcut answer to this question, but in general, it is worth implementing a business service when the use case corresponds to a workflow, during which multiple business entities need to be created or modified. In this case, the business service can be seen as a kind of coordinator. What is definitely not worth doing is to create a business service that exposes a Create Read Update Delete (CRUD) interface and simply forwards calls to one DAO service. Method names in business services should correspond to business operations: `registerStudent`, `sendRewardToGoodStudents` or `confirmOrder` are such examples.

5.2.3 *Database resources and connection pools*

When accessing a persistence store, the application needs to establish a connection with the database. One could imagine a scenario where the developer has to load a driver (provided by the database vendor), specify attributes (an IP address and a port, credentials, etc.), open a connection, send a query, receive a result and close the connection. Implemented naively, this scenario would raise two issues:

- The developer would need to decide where to define connection attributes. Of course, it would be a very bad idea to do that in the code. It would be better to store them in configuration files, but where? Should they be packaged in the `.war` file? All applications would need to answer this question, which means that a standard solution is valuable.
- The developer would also be responsible to manage resources very carefully and deal with non-functional aspects. Under heavy load, he should not open a connection for every request because this would likely starve the database server (which would anyways most likely refuse connection requests at a certain point. Here again, every application developer would need to find an answer to a common problem.

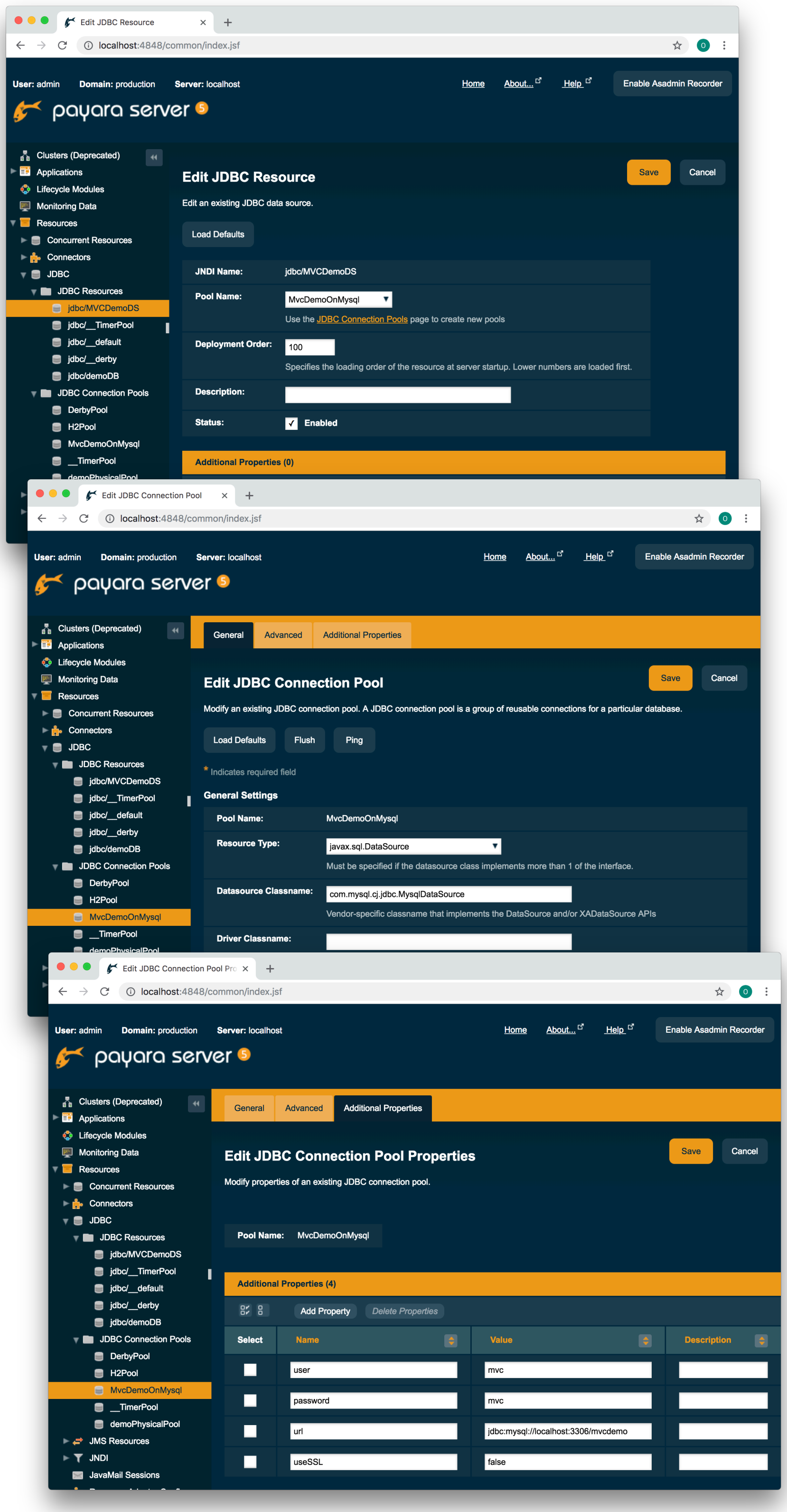
Java EE addresses these two issues by making a clear separation between the *development* and the *deployment* phase of an application.

The developers writes code that uses a logical connection named jdbc/myDatabase.

During the development phase, the developer works with *logical database connections*. In the previous chapter, we have introduced the concept of *dependency injection*. We have seen that when a servlet needs to invoke an EJB, it declares a dependency with the `@EJB` annotation. The application servers injects a reference to managed component at deployment time.

The sysadmin configures the application server: jdbc/myDatabase is mapped to a pool of physical connections to a concrete database server.

During the deployment phase, the IT ops engineer configures the application server, configures *physical database connections* and defines a mapping between logical and physical connections. This is when he specifies the connection parameters (IP address, port number, credentials, etc.). This is also when he configures how connections are to be pooled (so that they can be shared and reused by different clients).



Edit JDBC Resource

Edit an existing JDBC data source.

JNDI Name:	jdbc/MVCDemoDS
Pool Name:	MvcDemoOnMysql Use the JDBC Connection Pools page to create new pools
Deployment Order:	100 Specifies the loading order of the resource at server startup. Lower numbers are loaded first.
Description:	
Status:	✓ Enabled

Additional Properties (0)

Edit JDBC Connection Pool

Modify an existing JDBC connection pool. A JDBC connection pool is a group of reusable connections for a particular database.

* Indicates required field

General Settings

Pool Name:	MvcDemoOnMysql
Resource Type:	javax.sql.DataSource Must be specified if the datasource class implements more than 1 of the interface.
Datasource Classname:	com.mysql.cj.jdbc.MysqlDataSource Vendor-specific classname that implements the DataSource and/or XADataSource APIs
Driver Classname:	

Edit JDBC Connection Pool Properties

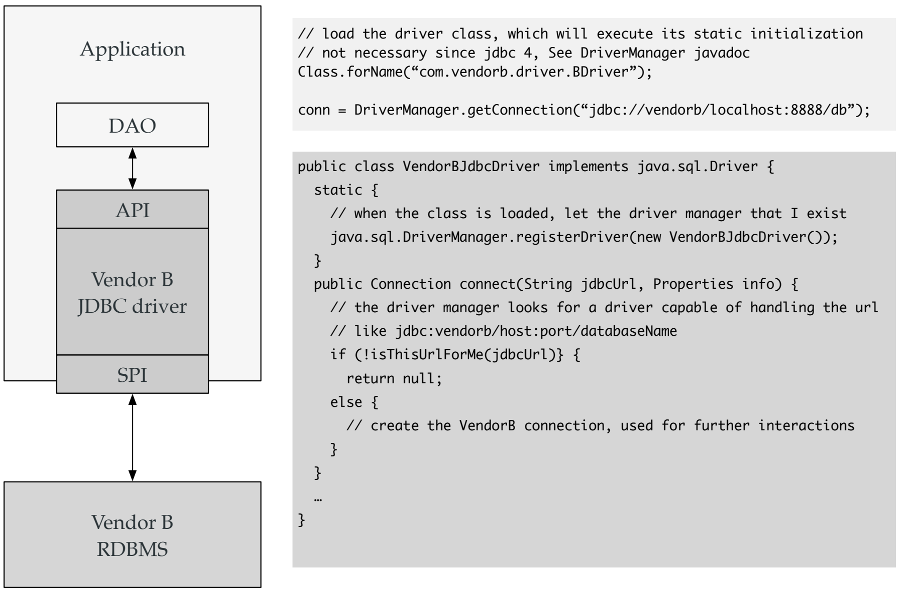
Modify properties of an existing JDBC connection pool.

Pool Name: MvcDemoOnMysql

Additional Properties (4)

Select	Name	Value	Description
☐	user	mvc	
☐	password	mvc	
☐	url	jdbc:mysql://localhost:3306/mvcdemo	
☐	useSSL	false	

Figure 5.2: The JDBC resource maps a logical name to a connection pool



```
// load the driver class, which will execute its static initialization
// not necessary since jdbc 4, See DriverManager javadoc
Class.forName("com.vendorb.driver.BDriver");

conn = DriverManager.getConnection("jdbc://vendorb/localhost:8888/db");
```

```
public class VendorBJdbcDriver implements java.sql.Driver {
  static {
    // when the class is loaded, let the driver manager that I exist
    java.sql.DriverManager.registerDriver(new VendorBJdbcDriver());
  }
  public Connection connect(String jdbcUrl, Properties info) {
    // the driver manager looks for a driver capable of handling the url
    // like jdbc:vendorb/host:port/databaseName
    if (!isThisUrlForMe(jdbcUrl)} {
      return null;
    else {
      // create the VendorB connection, used for further interactions
    }
  }
  …
}
```

Figure 5.3: JDBC provides an abstraction layer between the application and the database

5.2.4 *The Java DataBase Connectivity API*

JDBC is almost as old as Java: the first version of this API was part of the Java Development Kit 1.1 in 1997. The current version is JDBC 4.3 and is part of Java SE 9. JDBC was proposed to enable Java applications to interact with relational database management systems in a standard way. In other words, the goal was to allow developers to write data access logic once and to use it with databases from different vendors. In other words, the application should be able to store its data in a DB2 database, in an Oracle database or in MySQL database without any code change. To achieve this goal, JDBC defines both an Application Programming Interface (API) and a Service Provider API (SPI). The API is the interface used by application developers. It defines operations to open connections, to send queries, to receive results, etc. The SPI is the interface used by database vendors. It defines operations that have to be implemented in JDBC *drivers*.

Java EE applications do not use the `DriverManager` class. Instead, they use `DataSource` objects, which are injected in the services that contain data access logic.

Figure 5.3 shows how the JDBC driver provides an abstraction layer between the application code and a specific database management system. The application code only uses the API. Since all drivers have to implement it, the same application can interact with different applications, just by switching the driver. This raises an interesting question: how are drivers loaded and how does the application specify that it wants to use one particular driver? The code in the diagram describes what happens (note that the situation is a bit different since JDBC 4, but the general idea remains the same).

5.2.5 *Putting things together: implement a DAO with EJB and JDBC*

```
 1 @Stateless
 2 public class SensorJdbcDAO implements SensorDAOLocal {
 3
 4   @Resource(lookup = "jdbc/AMTDatabase")
 5   private DataSource dataSource;
 6
 7  public List<Sensor> findAll() {
 8     List<Sensor> result = new LinkedList<>();
 9     try {
10       Connection con = dataSource.getConnection();
11
12       PreparedStatement ps = con.prepareStatement("SELECT * FROM Sensors");
13       ResultSet rs = ps.executeQuery();
14
15       while (rs.next()) {
16         Sensor sensor = new Sensor();
17         sensor.setId(rs.getLong("ID"));
18         sensor.setDescription(rs.getString("DESCRIPTION"));
19         sensor.setType(rs.getString("TYPE"));
20         result.add(sensor);
21       }
22
23     } catch (SQLException ex) {
24       Logger.getLogger(SensorJdbcDAO.class.getName()).log(Level.SEVERE, null, ex);
25     } finally {
26       if (ps != null) { ps.close(); }
27       if (con != null) { con.close() };
28     }
29     return result;
30   }
31 }
```

Listing 5.1: Implement a DAO with a Stateless Session Bean

- Line 1: the DAO is a SLSB. This is particularly important from a transaction demarcation point of view. The container starts a transaction whenever a business method is called and commits it if the method returns a result. If the method throws an exception, then the container rollbacks the transaction.
- Lines 4-5: we inject a dependency on the data source. The container will assign a value to the dataSource variable. This will work only if the administrator has configured a JDBC connection in the application server.
- Line 7: we have not implemented a complete CRUD service, but only implemented a finder method.
- Line 10: we ask for a connection from the pool. Under heavy traffic, we might have to wait for one to become available (this will avoid that we overload the database server).
- Lines 12-13: we prepare a SQL query, send it to the server, receive a list of records.

- Lines 15-21: we iterate over the records and at each iteration, we create a POJO model object. We transfer data from the JDBC record to the model.
- Lines 23-24: we close the resources. Note that in reality, this will not close the connection but rather return it to the pool, so that it can be reused for another request.

5.3 OBJECT RELATIONAL MAPPING

TODO: content will be published later.

5.4 QUESTIONS

To answer these questions, you will need to have read the chapter but also to have done some research. Make sure that you are able to answer every question. Discuss your responses with your peers.

Part III

MODERN JAVA EE

This part presents a selection of frameworks that have been built on top of Java EE APIs and are currently very popular in the market. These frameworks aim to make developers more productive, by providing additional capabilities and by hiding some of the implementation details. They also address the evolution towards the cloud and micro-services architecture. This topic has been a gap in the Java EE specification (this will be one focus area for its future versions) for a long time, which is one reason to explain the popularity of the frameworks.

Part IV

SOFTWARE ENGINEERING PRACTICES

This part presents software engineering practices that are recommended when developing multi-tiered applications. The goal of these practices is to improve the productivity of the team and the quality of the developed product. A lot of these practices are related to testing, which is a broad subject. Both functional and non-functional requirements need to be tested. Different techniques and tools can be used for this purpose. Some of them are presented in the following chapters, with concrete case studies and examples.

6 LOAD TESTING AND PERFORMANCE MEASUREMENT

6.1 introduction

When creating a software product, the team has two consider two types of requirements:

- the *functional requirements* specify *what* the software should do. When doing a use case analysis, the team identifies the actors (the categories of users and external systems) which interact with the system. The team also enlists the features provided by the system and captures them as a list of use cases and scenarios.
- The *non-functional requirements* specify and quantify various qualities that the system should have. For instance, the team should assess the needs in terms of performance, scalability, availability, security, maintainability, manageability, etc. This is a critical part of the analysis, because it should drive the architecture of the system. Making a sound architectural decision is about making the right *tradeoff* between the cost of a solution and its impact on non-functional requirements. For instance, availability is always *somewhat* important. It is possible, but very complex and costly to build systems with a 99.999% uptime. Therefore, the first task is to define what is the appropriate availability level for the system. The second task is to propose a technical design that is expected to reach that level. The third task, very important and often overlooked, is to actually measure if the specified level is reached by the system. This has to be done through experiments and measurement.

When doing a use case analysis at the beginning of a project, always add an actor named "system administrator", or "IT Ops engineer". This will require the team to also look at the system from the point of view of people who run the system. This will encourage discussions about systemic qualities like reliability, scalability and security. Also, remember that every use case can be annotated with quantitative constraints (e.g. max response time, min throughput, etc.)

In this chapter we take a look at a particular non-functional aspect: how does the application behave under load? We explain how to setup experiments to measure performance, in terms of response time and throughput. We also show that testing an application under load can sometimes uncover bugs, particularly those related to concurrency. We present Apache JMeter, an open source tool that supports these testing activities.

An application that works when one developer tests it manually may break when 100 users access it at the same time.

6.2 testing web applications with apache jmeter

In the early 2000s, performing load tests on a web application required the use of expensive commercial tools. Apache JMeter was one of the first projects to offer an open source alternative. It has continuously evolved over time and remains a popular tool. Simply stated, JMeter allows you to define test scenarios, which are essentially sequences of HTTP requests (e.g. send a request to the login page, then a request to the account page, then a request to the products list page, etc.). When the scenario is ready, JMeter can simulate a large number of virtual users and thereby generate a load on the system. Measures are collected during this process and It is extensible and a number of plugins are available on the https://jmeter-plugins.org/ website. We recommend to install some of these plugins (in particular the plugins in the Thread Groups category), because they really improve the standard distribution.

6.2.1 *Installation*

Installing JMeter is very simple. An archive is available on the project website. The procedure to install plugins is described in the documentation. Once installed, JMeter is launched with a shell script located in the `bin` package.

Use the JMeter GUI to design and validate your automated tests. Disable the GUI to preserve resources when generating the real load.

The message displayed on the terminal, shown in Listing 6.1 is important. It makes it clear that JMeter should be used in two phases. During the test design phase, it is a good idea to use the GUI mode and to build the test scenarios with the interactive interface. However, once the test plan is ready, JMeter should be used in non-interactive mode to generate the load. The reason is that resources on the client machine should be used with care: the client should not be the bottleneck when running the test. How much load can be generated by JMeter, in other words how many concurrent users can be simulated, is not a trivial question. A special section in the documentation gives a few recommendations. Experts have also written articles on the topic and they are very useful when running a proper benchmark.

```
================================================================================
Don’t use GUI mode for load testing !, only for Test creation and Test debugging.
For load testing, use NON GUI Mode:
   jmeter -n -t [jmx file] -l [results file] -e -o [Path to web report folder]
& increase Java Heap to meet your test requirements:
   Modify current env variable HEAP="-Xms1g -Xmx1g -XX:MaxMetaspaceSize=256m" in
        the jmeter batch file
Check : https://jmeter.apache.org/usermanual/best-practices.html
================================================================================
```

Listing 6.1: Warning displayed when launching JMeter

6.2.2 *Concepts*

JMeter is used to create *test plans*. It is recommended to create different test plans for different parts or aspects of the application, to keep them small and manageable. A test plan is stored in a `.jmx` file and can loaded when running JMeter in non-interactive mode.

A test plan is a tree of elements and JMeter defines different types of elements:

- *Thread Groups* are used to simuate virtual users. There are different types of Thread Groups and it is important to understand how they work in order to interpret the test results.
- *Samplers* are used to generate individual requests. Initially, JMeter was used only to test web applications and provided an HTTP sampler. Today, it is possible to generate load for other protocols, such as FTP, LDAP, JDBC, JMS, etc.
- *Logic Controllers* are used to control the flow of a test: there are loop controllers, branching controllers, etc.
- *Listeners* are used to monitor the activity of samplers and to compute statistics, present information in tables or generate graphs. Listeners are very useful when *designing* a test plan, but some of them consume a lot of resources (CPU, RAM). For this reason, they should be disabled during the *execution* of the test plan.
- *Timers* can be used to pause the virtual user. One reason to do this is to generate more realistic test loads, as human users need *think time* between actions. Another reason is to manage resources (on the server and client side). Timers are applied before each sampler in its scope.
- *Configuration Elements* are used to setup samplers. For instance, the management of HTTP cookies and headers is done via configuration elements.
- *Pre- and Post-Processor Elements* can be used to execute an action before a request is sent, respectively after a response has been received. For instance, a post-processor can parse the response and extract some value in a variable. This variable can be used to prepare the next request.

6.2.3 *Special features*

JMeter can be used on a single client machine and this is often enough to study the behavior of an application under load. However, it is also possible to generate a bigger load, by running a test with multiple client machines. In this case, a master node is setup to aggregate all test results. The documentation contains a specific section on this topic.

Test plans can be created from scratch, using the tree view in the JMeter user interface. It is also possible to create a test plan by recording browser activity. To do that, JMeter is first configured to operate as an HTTP proxy. The browser is configured to go via through that proxy. The user then goes through the scenario, clicks on links, fills out forms, etc. Hence, JMeter senses every HTTP request and can automatically generate a test plan, with HTTP samplers. The generated test plan is a good starting point, but often needs to be cleaned up and adjusted. This technique is described in a specific tutorial in the JMeter documentation.

6.3 CASE STUDY: THREAD-SAFETY OF THE SERVLET API

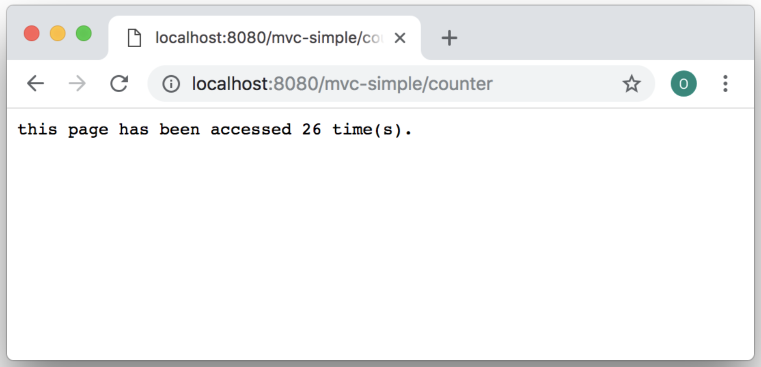
To demonstrate the use of JMeter and to show how load testing can uncover bugs, we now consider a concrete use case:

1. We first describe what the application is supposed to do and present an incorrect implementation that passes manual tests.
2. We then explain why the implementation is incorrect and discuss concurrency issues with servlets.
3. We then show to *create* and *run* a test plan with JMeter.
4. We interpret the results and prove that the code is incorrect.
5. Finally, we fix the problem, re-run the test and show that the behavior is now correct.

6.3.1 *Requirements: implement a shared access counter with a servlet*

The goal is to create a simple web application, which manages a counter. The user interface is shown in 6.1. Whenever a user accesses the URL, the counter is incremented and the page displays its current value (This page has been access *n* times). The counter is global, i.e. it is shared by all users and not bound to HTTP sessions.

Implementing this feature looks easy enough. Let us create an HttpServlet and define a `counter` instance variable. In the `doGet` method, simply get and increment its value and generate the response page. Let us also make it possible to reset the counter by sending a `command` parameter in the query string with the `reset` value. The code is shown in 6.3.


localhost:8080/mvc-simple/co
localhost:8080/mvc-simple/counter

```
this page has been accessed 26 time(s).
```

Figure 6.1: The application manages an access counter shared by all users

```java
package ch.heigvd.amt.mvcsimple.presentation;

import ch.heigvd.amt.mvcsimple.model.Quote;

import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;

@WebServlet(name = "CounterServlet", urlPatterns = "/counter")
public class CounterServlet extends HttpServlet {

    private int counter = 0;

    protected void doGet(HttpServletRequest request, HttpServletResponse response)
          throws ServletException, IOException {
        String command = request.getParameter("command");
        if ("reset".equals(command)) {
            counter = 0;
        } else {
            counter = counter + 1;
        }
        response.getWriter().println("this page has been accessed " + counter + "
             time(s).");
    }
}
```

Listing 6.2: Incorrect implementation of the counter servlet

6.3.2 *Problem: servlets are not thread-safe*

Using member variables in servlets is dangerous. Either they have to be thread-safe objects or the access need to be synchronized.

At first glance, the application works as expected. When the developer deploys the application and performs manual tests in the browser, the counter seems to work perfectly. Time to release to production? Not quite: there is a serious problem with this code. When using a development platform such as Java EE, the lifecycle of objects is managed by a framework. To write correct code, it is important to study what happens and to be aware which methods are thread-safe and which are not.

The Servlet specification clearly states that it is possible for several threads to be in the `doGet` method at the same time. Therefore, it is possible to have a race condition on instance variables. The Javadoc documentation for the `HttpServlet` class states: *Servlets typically run on multithreaded servers, so be aware that a servlet must handle concurrent requests and be careful to synchronize access to shared resources. Shared resources include in-memory data such as instance or class variables and external objects such as files, database connections, and network connections.*

To show that the code in Listing **??**, let us consider a concrete scenario. Let us imagine that two browsers send a request to the servlet at the same time. As a result, the `doGet` method is executed twice, on two threads `T1` and `T2`. Let us imagine that before the scenario, the counter has a value of `100`. If the two threads reach line 22 at the same time, the following sequence of operations is possible:

1. `T1` reads `counter` value (`100`)
2. `T2` reads `counter` value (`100`)
3. `T1` sets `counter` value to `100 + 1` (`101`)
4. `T2` sets `counter` value to `100 + 1` (`101`)

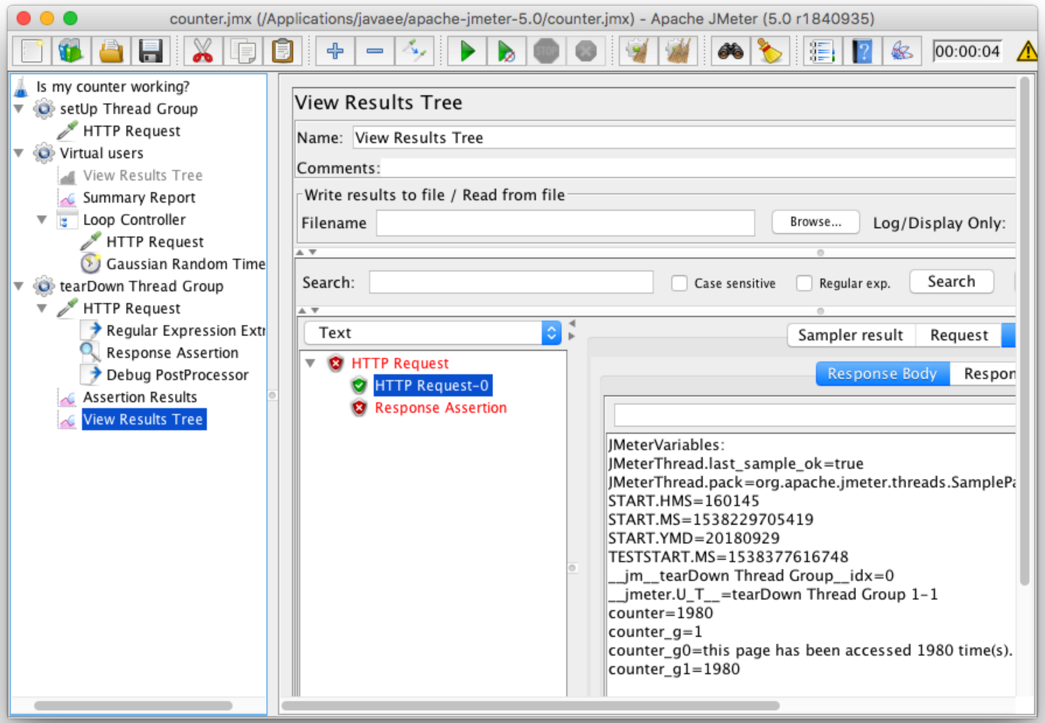
This scenarios that it is possible to miss counter updates. Hence, if the servlet is accessed 1'000 times under load, it is possible that the counter has a smaller value than 1'000. How do we show that this is indeed possible? How do we create an experiment to produce the bug and have an evidence of the problem? Working with the debugger might provide some help, but we would most likely have to modify the code and introduce some statements to pause the execution of the program. Let us take a different route and create a race condition through load testing.

6.3.3 *Design of a test plan with JMeter*

A demonstration has been recorded in a webcast. It is available in the companion Youtube playlist, in a series of 4 videos entitled "Load testing with JMeter"

The test plan created in JMeter to produce the error condition is shown in Figure 6.2. The tree view on the left contains the following elements:

- The *setUp Thread Group* is used to send one single HTTP request at the beginning of the test procedure.
- To do that, we configure the HTTP Request sampler to send a `GET` request to `/mvc-simple/counter?command=reset`.
- *Virtual users* is a thread group that simulates concurrent users. We set the number of threads to 20 and do 1 iteration of the plan.



```
JMeterVariables:
JMeterThread.last_sample_ok=true
JMeterThread.pack=org.apache.jmeter.threads.SamplePa
START.HMS=160145
START.MS=1538229705419
START.YMD=20180929
TESTSTART.MS=1538377616748
__jm__tearDown Thread Group__idx=0
__jmeter.U_T__=tearDown Thread Group 1-1
counter=1980
counter_g=1
counter_g0=this page has been accessed 1980 time(s).
counter_g1=1980
```

Figure 6.2: The test plan defines a complete scenario

- the View Results Tree is a listener that allows us to see that last requests and responses processed by JMeter. In the screenshot, it is disabled because we have completed the test design phase and do not need it anymore.
- In the *Loop Controller*, we specify that we want every virtual user to perform 100 iterations of the scenario.
- The scenario consists of sending an HTTP request to `/mvc-simple/counter` and to pause for a brief period (the *Gaussian Random Timer* is configured with a deviation of 10 ms and delay offset of 30 ms.
- The *tearDown Thread Group* is used to issue one last HTTP request to retrieve the value of the counter at the end of the test. We use the *Regular Expression Extractor* to retrieve the counter value, the Response Assertion to check if the counter value is equal to 2001 (*20 virtual users * 100 iterations per virtual user + 1 last request*).
- the *Debug PostProcessor*, *Assertion Results* and *View Results Tree* are used to display the test results and to inspect the value of the extracted variables.

6.3.4 *Interpretation of the results*

In the screenshot of the JMeter test plan, we see that the *Response Assertion* is displayed in red, which indicates a problem: the expect final value for the counter is not 2001. If we click on the *HTTP Request-0* node in the tree view, we see the evaluation of the regular expression and the value of the counter variable. It is equal to 1980. When running the load test multiple times, counter will generally have a different value at each run, because concurrency makes the behavior non deterministic.

6.3.5 *Correction of the bug*

The `CounterServlet` class can easily be fixed, as shown in Listing 6.3. The code that accesses and modifies the member variable is wrapped in a synchronized block (lines 11-13). An alternative would have been to use a thread-safe class (`AtomicInteger`) instead of the int primitive type for the counter variable.

```
1  @WebServlet(name = "CounterServlet", urlPatterns = "/counter")
2  public class CounterServlet extends HttpServlet {
3
4      private int counter = 0;
5
6      protected void doGet(HttpServletRequest request, HttpServletResponse response)
           throws ServletException, IOException {
7          String command = request.getParameter("command");
8          if ("reset".equals(command)) {
9              counter = 0;
10         } else {
11             synchronized (this) {
12                 counter = counter + 1;
13             }
14         }
15         response.getWriter().println("this page has been accessed " + counter + "
                time(s).");
16     }
17 }
```

Listing 6.3: The counter servlet modified to deal with concurrent requests

6.4 CASE STUDY: HTTP SESSIONS AND SCALABILITY

While HTTP is a stateless protocol, many web applications built on top of the protocol have a notion of session. Think about applications with *login* and *logout* features. Think about a typical e-commerce site with a *shopping cart*. In these applications, there is a state associated with every session and it needs to be stored somewhere. As a matter of fact, there are quite a few places to store it and the choice can have a dramatic impact on performance, scalability and availability. There is no single best practice and trends have evolved over the years.

When Java EE appeared on the market, the most common practice was to avoid storing session state in the database. The reason was that the database was usually on a remote host. At the time, networks were slower and even if the application host and database host were on the same internal network, remote calls were slow. Memory was also expensive, which means that databases were not able to store large data sets in memory caches. For these reasons, it made sense to store conversation state in the presentation tier.

The servlet API makes this very easy, thanks to the `HttpSession` class. When implementing the `doGet` method, the developer can obtain a HttpSession from the request parameter. He can then store arbitrary object graphs by calling `session.setAttribute("name", value)`. The great thing about this API is that it is very easy to use. The terrible thing about it is that it is very easy to use and that inexperienced developers sometimes do not realize the implications of making these calls. Obviously, storing an object in the session consumes memory. This memory remains allocated as long as the session is active. There are two main reasons for the session to become inactive. Sometimes, the developer can explicitly call `session.invalidate()` when he knows that the user has finished his session (e.g. he has clicked on a *logout* button). However, users often just stop interacting with the web application and it is impossible to know for sure that they are done. Application servers can be configured with a maximum session idle period: if no request is received from a user within this period, then the session is terminated and the resources are reclaimed. Setting the maximum session idle time to 5 minutes, 30 minutes or 3 hours can make a huge difference on a high-traffic web application.

Another factor to consider is the JVM configuration parameters for the application server. When using a Java EE application server in development, one often goes with the default parameters. This includes memory settings, such as the maximum heap size. In production, it is usually necessary to increase the default value. By how much? If you know how many active sessions should be supported by a node and if you know how much memory is consumed by the session state, then you can estimate the memory requirements for this aspect of the application. Of course, while this is something that you can model and calculate, it is also something that you should measure through experiments. As you can imagine, this is where a tool like JMeter is very helpful.

[TODO: document test procedure with JMeter]

6.5 case study: performance impact of pagination

[TODO]

6.6 CASE STUDY: PERFORMANCE IMPACT OF CACHING

[TODO]

6.7 CASE STUDY: PERFORMANCE IMPACT OF ORM TUNING

[TODO]

6.8 QUESTIONS

To answer these questions, you will need to have read the chapter but also to have done some research. Make sure that you are able to answer every question. Discuss your responses with your peers.

[TODO]