The enterprise Java framework was first released in 1999. It has undergone a great number of improvements over the last decades, but the learning curve is still very steep. So in this article, we're going to take an in-depth look at version 8 of the Java EE framework.

The two most popular Java frameworks for server-side application development are **Java EE** (*Enterprise Edition*) and **Spring**. Java EE is the *official* specification, whereas Spring (unofficially) describes pretty much the same functionality, but in many cases does so in a slightly different, often easier way.

(Later on, we'll go into greater detail about what the Java EE specification *is*, and what the term actually means.)

Should you use Java EE for your next projects, then? No, probably not. Because what's important to know about Java EE, is that it's a comparatively heavy framework, in terms of both setup and actually running it (in production). The Java EE framework has greatly improved over the last decade, but in terms of documentation, ease-of-use and operational simplicity, it still doesn't match up against **Spring Boot**.

Becoming knowledgeable in Java EE can still be worthwile, though. For example, if you're looking to expand your software consultancy services to enterprise clients. The *EE* part of Java EE (Enterprise Edition) is still very fitting, as the framework is still almost exclusively used by the enterprises and BigCo's like banks, insurance companies, government agencies, S&P500, AEX, etc.

Table of Contents

Java EE Theory

Java EE is a Java framework for developing server-side applications. Among other things, it allows you to:

- create web APIs by accepting incoming HTTP requests
- save and retrieve data from databases
- send and receive messages from message brokers
- connect to other backend systems via HTTP or SOAP
- package applications into single units of deployment (.war files)

The framework is very much geared towards enterprises and big corporations. As such, it often ends up being popular in the domains of banking, insurance, accounting, finance, and manufacturing.

Like the programming language Java itself, Java EE used to be owned and managed by **Oracle**. However, between 2017 and 2019, Oracle has transfered the ownership of the Java EE framework to the **Eclipse Foundation**, resulting in the framework being renamed *Jakarta EE* from version 8-9 onward.

Framework Name	Version	Year of Release
J2EE	1.2	1999
J2EE	1.3	2001

Framework Name	Version	Year of Release
J2EE	1.4	2003
Java EE	5	2006
Java EE	6	2009
Java EE	7	2013
Java EE	8	2017
Jakarta EE	8	2019
Jakarta EE	9	2020

It's going to take time for the term Jakarta EE to really catch on, as the term Java EE is still very much ingrained in the corporate software industry.

We will be setting up a **Java EE 8** application in the build-along section of this article.

Specifications and APIs

You should think of Java EE as a collection of *recipes*. Recipes for typical pieces of functionality within a server application. The technical term for a Java EE recipe is a *Java Specification Request* (JSR), and each JSR describes a single unit of functionality within the Java EE framework. For example:

- Creating a RESTful web API? **JSR 370**
- Validating incoming HTTP request bodies? **JSR 380**
- Connecting and reading (or writing) from a database? **JSR 338**
- Publishing messages to a message broker? **JSR 343**

This modular setup lets developers pick and choose which parts of the framework they want to use.

In an actual project, this typically means that your application has a single Maven dependency on the full Java EE API.

And what's interesting to understand, is that these Java EE API modules consist of nothing but interfaces, annotations and simple POJOs. *There is no actual Java EE framework functionality in* ***any*** *of these API modules.* The *real* implementation of the Java EE specifications, interfaces and annotations lies with the host environment in which your Java application(s) will be running. And such host environments are referred to as **Java EE application servers**, or simply **(application) servers**.

Application Servers and Implementations

Java EE applications can't stand on their own—they require a *Java EE application server* to run in, because the server provides the application with the facilities it needs to carry out its tasks.

This ties back to the story about JSRs (Java Specification Requests) from earlier. Since your application only depends on Java EE **API** modules, it's agnostic of the actual implementation of these modules (at least in theory

😅). It's up to the application server to provide the actual (runtime) **implementations** to these API modules.

The image above should make clear that Java EE application servers are to be thought of as facilitating environments, in which multiple (unrelated) Java apps from different teams or departments can simaltuneously run alongside each other, each application pseudo-isolated from its neighbours.

An application server doesn't have to implement every single Java EE specification, though. Different servers can have varying degrees of specification *compliance*.

For example, the **Apache Tomcat** server only implements the web layer of the Java EE specification and, as such, is referred to as a **web server**. The **JBoss WildFly** server, on the other hand, implements every part of the Java EE specification, and can therefore be referred to as a **(Full) Java EE application server**.

Below's an overview of the most popular Java EE servers.

Application Server	Spec. Compliance
Apache Tomcat	Web-only
Eclipse Jetty	Web-only
JBoss WildFly	Full
IBM WebSphere	Full

Application Server	Spec. Compliance
Oracle WebLogic	Full
Apache TomEE	Full

When deploying a Java EE application to some server, there will be lots of additional libraries available on the classpath at runtime. These are all *implementation libraries* for the different Java EE API modules which the application might depend on. So there's a big difference between *compile-time* API libraries and *runtime* implementation libraries.

For example, by having a Maven *compile* dependency on the (full) **Java EE API** library, you'll be able to decorate your classes and methods with the `@Path` and `@GET` annotations (part of JSR 370) and create a simple HTTP endpoint like this:

```
@Path("/ping")
public class PingResource {

    @GET
    public Response ping() {
        return Response.ok("Ping").build();
    }

}
```

In and of themselves, Java runtime annotations don't do anything. That's why the application server has an *implementation library* for JSR 370, which it places on the classpath during runtime. This implementation would then have a built-in mechanism which scans your application for all classes with the `@Path` annotation, and register them as HTTP endpoints.

Server developers typically choose to implement most of the **500+** JSRs themselves. Often, these implementation are then packaged into standalone libraries so that, in theory, they can be re-used by other server developers.

As such, there are typically only 1 or 2 popular implementation libraries for each specification, which may or may not find their way into the standard configuration of multiple different application servers over time.

Specification	Popular Implementation Libraries
JSR 370 (JAX-RS)	RestEasy, Jersey
JSR 365 (CDI)	Weld, OpenWebBeans
JSR 338 (JPA)	Hibernate, EclipseLink
JSR 907 (JTA)	JBossTS (Narayana), Atomikos

JBoss WildFly

JBoss used to be the name of a company which has developed one of the most popular Java EE application servers in the world: **JBoss EAP** (Enterprise Application Platform). The other application server which JBoss is known for, is called **WildFly**. The difference? Technically, nothing. WildFly is the free community edition and JBoss EAP is the paid enterprise edition (with official support) of the *same* application server.

As a sidenote, JBoss actually got acquired by **RedHat** in 2006. And then, in 2019, RedHat got acquired by **IBM**. So the JBoss EAP / WildFly server is technically part of IBM's portfolio now, together with IBM's self-developed Java EE application server called **WebSphere**.

We'll be working with the **JBoss WildFly** application server (version **20**) in the build-along section of this article.

Project Overview

We'll be working on a Java EE application with the following features:

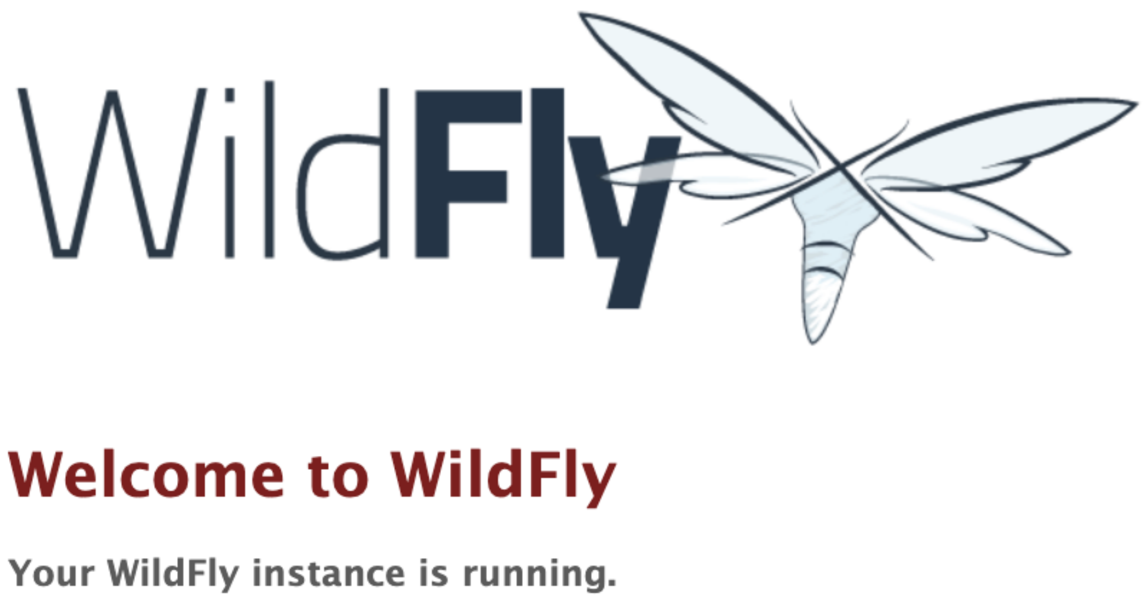
- customers can sign up via the `POST /accounts` endpoint
    - their `name` and `email` are saved into the database
    - they're sent a welcome e-mail
- all accounts can be queried via the `GET /accounts` endpoint
- a single account can be queried via the `GET /accounts/{id}` endpoint
-
- **Java 11** - newer language versions aren't supported by the WildFly version we're going to pick later
- **Maven**
- **Docker Compose** - for running the database locally
- **IntelliJ**

WildFly Setup

Start out by downloading the **JBoss WildFly 20.0.0.Final** application server (Java EE Full & Web Distribution) from the official **downloads page**. Once it's downloaded, unzip the the archive to a convenient location or simply leave it inside your `~/Downloads/` folder (which is what I will do).

I shall refer to the WildFly root directory as `$JBOSS_HOME` from now on, which would resolve to `~/Downloads/wildfly-20.0.0.Final/` on my system.

To *start* this server, simply `cd` into the `$JBOSS_HOME/bin/` folder and run the `./standalone.sh` script. After a few seconds, you should be able to visit **localhost:8080** and be greeted with the following welcome screen.


Welcome to WildFly

Your WildFly instance is running.

(To stop the server, simply press Cmd + C or Ctrl + C from the terminal window where the `./standalone.sh` script is running.)

WildFly also comes with a useful management dashboard, available at **localhost:9990**, from where we'll be able to view runtime stats about WildFly itself and all Java EE applications deployed to it. In order to access the dashboard, we'll have to create a management user first, though. To do so, `cd` into `$JBOSS_HOME/bin/` and run the `./add-user.sh` script.

```
> Type of user?
management

> Username?
jessy

> Password?
********

> About to add user 'jessy' for 'ManagementRealm' - is this correct?
yes

> Is this new user going to be used for one AS process to ... ?
no
```

You should now be able to log in and click around in the management dashboard (if the server is running).

Hello World

Let's start out by creating a POM file for our new Maven project. You can either set it up from scratch, or follow **this example**. At the very least, the POM should have a *provided* dependency on the **Java EE API** and a *compile* dependency on the **SLF4J API** (used for logging). Additionally, I recommend including the very useful **Lombok** library.

Next, create a folder called `src/main/webapp/` where all static resources like HTML, CSS and JavaScript files are to be placed. Add an **index.html** file here, containing a simple `<h1>Hello World</h1>` header element.

We're almost ready to serve this HTML page - all that's left to do, is:

- whitelisting the `index.html` file, so it can be accessed without authentication
- configuring our application's **context root** - *remember that multiple applications can run alongside eachother within the same application server, so some kind of server-wide unique URL namespace is needed for each application to place their HTTP endpoints under (this is the context root)*

For this, we'll have to create some *deployment descriptors*, i.e.: XML config files with information about access management and deployment preferences.

Deployment descriptors for *web* applications are to be placed inside the `src/main/webapp/WEB-INF/` folder. So create this folder, and add the following two files to it:

- **web.xml** for whitelisting the `index.html` file
- **jboss-web.xml** for configuring the application's context root

By setting the context root to `/store`, all of our application's HTTP endpoints will be available under `http://localhost:8080/store/**`. Note that you're also allowed to set the context root to `/`.

After deploying this minimal application to WildFly in the next section, we should be greeted with the "Hello World" H1-header when visiting `http://localhost:8080/store`.

Command Line Deployment

To deploy our application to WildFly, we first need to *package* it. This is done via the **Maven WAR plugin**, so simply add it to the **plugins** section of your POM file.

Next, run the `mvn package` command to generate a .war file (more on deployment artifacts later). Depending on the Maven artifact ID and version of your application (`store` and `1.0.0` in my case), the .war file can be found inside the application's `target/` directory as `store-1.0.0.war`. Make note of the exact path.

From this point, deploying our application is relatively straightforward. After making sure that the WildFly server is running via the `./standalone.sh` script, simply `cd` into the same `$JBOSS_HOME/bin/` folder from a different terminal window and execute the `./jboss-cli.sh` script with the following arguments (adjust with the actual path of your .war file):

```
./jboss-cli.sh -c --command="deploy ~/app/target/store-1.0.0.war --force"
```

(The `--force` flag is used for overwriting any previously deployed version of the application, making this command suitable for both new deployments and redeployments of the same application.)

If the command completes without error, and if you spot a message like `WFLYSRV0010: Deployed "store-1.0.0.war"` inside the terminal window where the `./standalone.sh` script is running, then the deployment has succeeded.

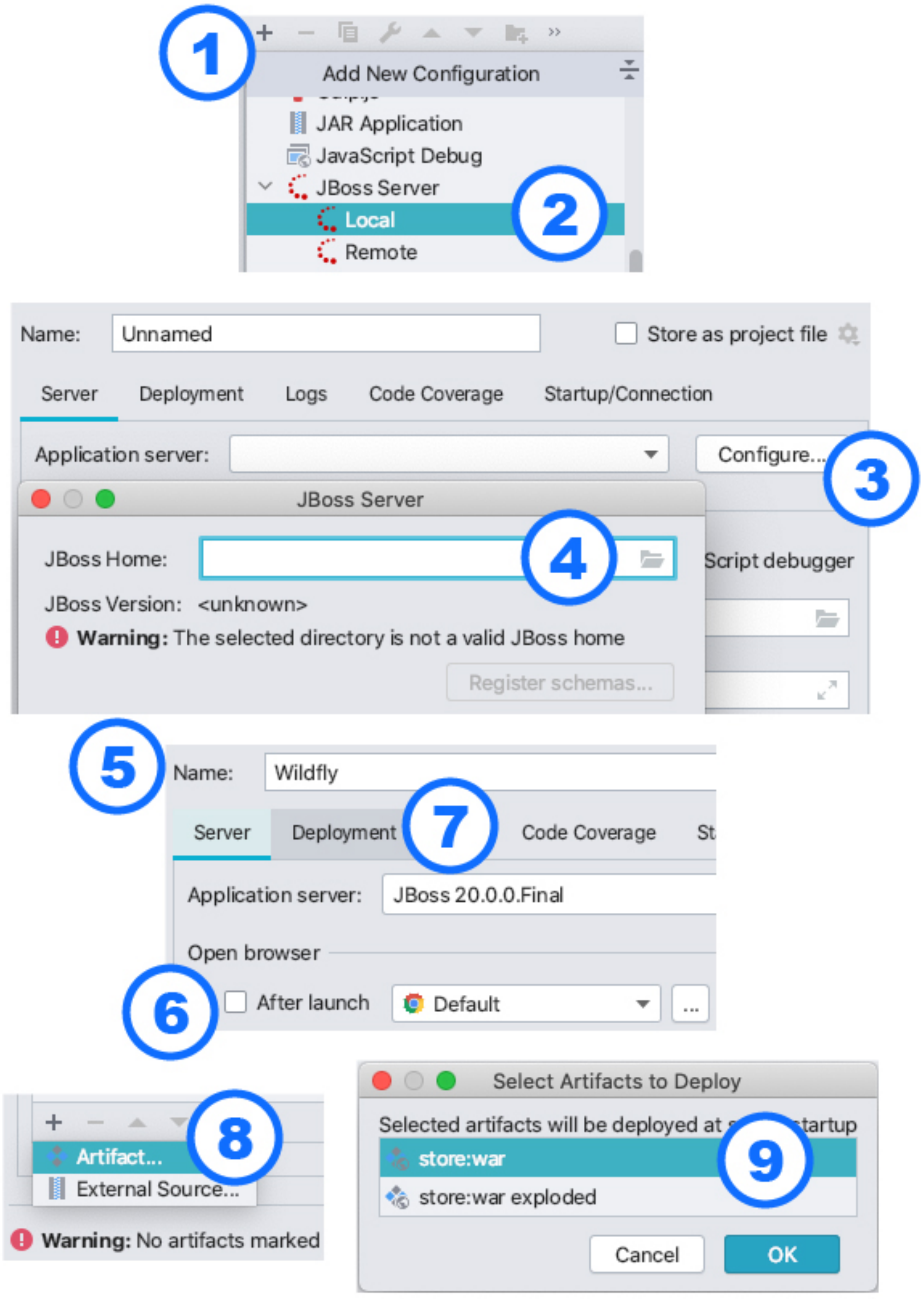
If you visit **localhost:8080/store** (where `/store` refers to the context root we've configured earlier), then WildFly should serve the application's `index.html`.

IntelliJ Deployment

Command line deployments are useful because they can be fully automated, which is what you need in production, acceptance and integration environments. For local development, it's much easier to deploy directly from your IDE, though.

After importing our Java EE application into IntelliJ, check out the screenshots below on how to create a Run/Debug configuration for it:

1. Create a new Run/Debug configuration via the menu **Run** -> **Edit Configurations...** and click the little plus (+) sign
2. Choose **JBoss Server (Local)**
3. Click the **Configure...** button to configure the application server
4. Click the directory icon to select your unzipped `$JBOSS_HOME` WildFly distribution (e.g.: `~/Downloads/wildfly-x.0.0.Final/`)
5. Specify a name for the Run/Debug configuration (optional)
6. Untick the open-browser-after-launch box (optional)
7. Open the **Deployment** tab
8. In the bottom-left corner, click the plus (+) sign and click **Artifact...** to select a deployment artifact
9. Select the (non-exploded) WAR module of your application and click **OK**



Name: Unnamed

Application server:

JBoss Home:

JBoss Version: <unknown>

Warning: The selected directory is not a valid JBoss home

Name: Wildfly

Application server: JBoss 20.0.0.Final

Default

Warning: No artifacts marked

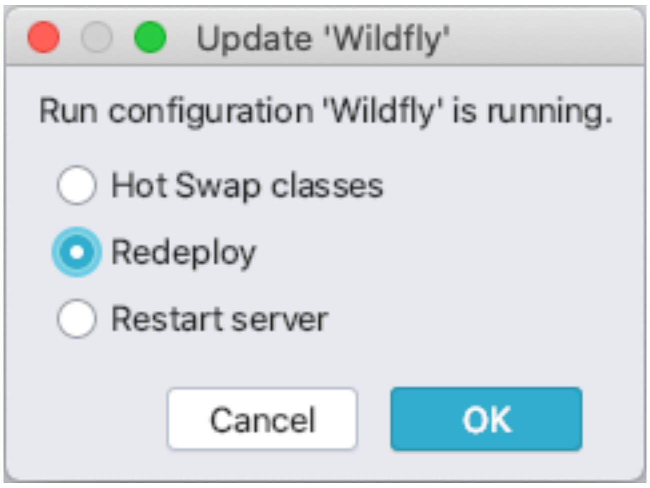
Selected artifacts will be deployed at server startup

store:war

store:war exploded

Next, click **Apply** and **Run/Debug** to start the server and deploy your application to it via IntelliJ. Also, make sure that the server is not already running via the `./standalone.sh` script in some terminal window, or this won't work. If there aren't any errors, you should be greeted with the same "Hello World" welcome screen when visiting localhost:8080/store.

Whenever you've made some code change in IntelliJ, and try to re-execute the Run or Debug configuration, you might be prompted with the following dialog.


Update 'Wildfly'

Run configuration 'Wildfly' is running.

( ) Hot Swap classes
(•) Redeploy
( ) Restart server

For 99% of changes, you can simply pick **Redeploy**, which is faster than restarting the entire WildFly server. I only recommend trying **Restart server** if WildFly fails to pick up your changes for whatever reason.

Deployment Artifacts (.jar, .war, .ear)

Not every Java EE application is necessarily a *web* application. Certain types of applications like scheduling systems or financial reporting systems might not have an HTTP interface at all, and only accept requests via some message queue. In Java EE terminology, such systems are referred to as **backend**.

If the reporting system also comes with some kind of web-based admin dashboard for the finance department, then the dashboard component of the system is referred to as **frontend**. This frontend component would also be the module which defines the `web.xml` and `jboss-web.xml` deployment descriptors inside the `src/main/webapp/WEB-INF/` folder, so it can set the context root for its URLs.

In a traditional Java EE setup, backend components (business logic) would be packaged as **.jar** artifacts (Java Archives) and frontend components (web interfaces and view logic) would be packaged as **.war** artifacts (Web Application Archives). These .jar and .war artifacts would then be grouped together into a single **.ear** artifact (Enterprise Application Archive), containing both the application's frontend and backend components. This single .ear artifact would then be deployed to some Java EE application server.

Quick sidenote: it's technically possible to package frontend and backend components into a single **.war** file as well (more on this later), but that hasn't been the traditional setup.

So what's the deal with this overly complex setup?

It's originally meant as a way for the server to isolate the frontend from the backend at a classloader-level. At runtime, the application server allows frontend classes (from .war artifacts) to access and invoke backend classes (from .jar artifacts), but not the other way around. Because—as the old saying goes—business logic isn't allowed to depend on view logic.

Nevertheless, I recommend only ever using the **.war** artifact for Java EE application deployments.

Proper frontend/backend isolation can be achieved just as easily with a proper multi-module Maven setup. Plus, the operational complexity is reduced significantly if your Java EE application gets packaged into a single uber-war file containing all of its frontend modules, backend modules and external libraries. There's also this myth that many Java EE features can't be used from a Web Application Archive, but this isn't true; pretty much all Java EE technologies can be used from a .war artifact, including JAX-RS, CDI, EJBs, JPA and JMS (more on these later).

REST Endpoints (JAX-RS)

JAX-RS is an abbreviation for the *Java API for RESTful Web Services* (officially known as JSR 370). It's a specification which lets you create HTTP endpoints via annotations like `@GET` and `@POST`.

To get started, we need a class which defines the JAX-RS *application path*. Start out by defining a sensible root package for the build-along project (in my case: `com.jessym.store`), and add a class **like this**:

```
package com.jessym.store;

import javax.ws.rs.ApplicationPath;
import javax.ws.rs.core.Application;

@ApplicationPath("/api")
public class WebApplication extends Application {
}
```

The `@ApplicationPath` annotation defines a common prefix for all JAX-RS HTTP endpoints within the application, but keep in mind that the context root always comes first. So static resources like `index.html` are served at `http://localhost:8080/store/**`, and JAX-RS HTTP endpoints are served at `http://localhost:8080/store/api/**`.

In Java EE terminology, HTTP endpoint collections are usually referred to as *resources*, so let's create a `resources` package with the following `PingResource` class:

```java
package com.jessym.store.resources;

import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.core.Response;

@Path("/ping")
public class PingResource {

    @GET
    public Response ping() {
        return Response.ok(hashCode()).build();
    }

}
```

Next, redeploy the updated application to WildFly (either manually or via IntelliJ) and visit localhost:8080/store/api/ping. If you're greeted with a random Java hash number, as opposed to a 404 error message, it means that WildFly has successfully picked up this first JAX-RS endpoint. (You might also notice that you're getting a new hash code on every refresh—more on this later.)

Let's set up an `AccountResource` in a similar fashion:

```java
package com.jessym.store.resources;

import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.core.Response;

@Path("/accounts")
public class AccountResource {

    @GET
    public Response list() {
        return Response.ok("jessy, bob, alice").build();
    }

    @GET
    @Path("/{id}")
    public Response findById(@PathParam("id") Long id) {
        return Response.ok(id).build();
    }
}
```

In the first `list` endpoint (/accounts), we're just returning a string with some account names in the HTTP response.

In the second `findById` endpoint (/accounts/{id}), we're extracting the `{id}` as a `Long` variable from the URL, and echoing this back in the HTTP response.

JSON Binding

Instead of returning numbers and strings from our HTTP endpoint methods, we can also return POJOs for JAX-RS to automatically serialize into JSON objects.

Let's add a `model` package to our project, and let's add the following `Account` POJO class to it.

```java
package com.jessym.store.model;

import lombok.Data;

@Data
public class Account {

    private Long id;
    private String name;
    private String email;

    // I'm using Lombok's @Data annotation to
    // automatically generate getters and setters

}
```

A particular instance of this class might be serialized into the following JSON:

```json
{
  "id": 1,
  "name": "jessy",
  "email": "jessy@example.com"
}
```

For this, we have to let the JAX-RS implementation library know about the *content type* of our HTTP responses. This is done via the `@javax.ws.rs.Produces` annotation, which we can use to not only serialize outgoing POJOs into JSON, but also automatically set the `Content-Type` HTTP response header to `application/json`.

Below's an example of what this would look like.

```java
package com.jessym.store.resources;

import com.jessym.store.model.Account;

import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;
import java.util.List;

@Path("/accounts")
@Produces(MediaType.APPLICATION_JSON)
public class AccountResource {

    @GET
    public List<Account> list() {
        Account account = new Account();
        account.setId(1L);
        account.setName("bob");
        account.setEmail("bob@example.com");
        return List.of(account);
    }

    @GET
    @Path("/{id}")
    public Account findById(@PathParam("id") Long id) {
        Account account = new Account();
        account.setId(id);
        account.setName("alice");
        account.setEmail("alice@example.com");
        return account;
    }
}
```

(Note that the `@Produces` annotation can be used at both the class level and the method level.)

After deploying these changes to WildFly, you should find that the **/accounts** and **/accounts/{id}** endpoints both return a piece of JSON in their HTTP response.

```
{"id":42,"name":"alice","email":"alice@example.com"}
```

Please note that, as a sensible default, the JAX-RS implementation library automatically uses the POJO's field name as the JSON field name during serialization. In order to override this default, simply add the `@JsonbProperty` annotation to the desired field, which will change the resulting JSON field name from `id` to `ID` in the example below.

```java
@Data
public class Account {

    @JsonbProperty("ID")
    private Long id;

    // ...

}
```

In addition to *returning* JSON in HTTP responses, we can also accept *incoming* JSON payloads from HTTP requests. For this, we will use the `@javax.ws.rs.Consumes` annotation (the counterpart of `@Produces`). Keep in mind that you can't accept request payloads with (ordinary) GET requests, only with POST, PATCH, PUT and DELETE requests.

```java
package com.jessym.store.resources;

import com.jessym.store.model.Account;
import com.jessym.store.resources.dto.RegisterAccountRequest;

import javax.validation.Valid;
import javax.ws.rs.Consumes;
import javax.ws.rs.POST;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;

@Path("/accounts")
@Produces(MediaType.APPLICATION_JSON)
@Consumes(MediaType.APPLICATION_JSON)
public class AccountResource {

    @POST
    public Account register(RegisterAccountRequest request) {
        Account account = new Account();
        account.setId(1L);
        account.setName(request.getName());
        account.setEmail(request.getEmail());
        return account;
    }

    // ...

}
```

Here, `RegisterAccountRequest` refers to the following POJO:

```java
package com.jessym.store.resources.dto;

import lombok.Data;

@Data
public class RegisterAccountRequest {

    private String name;
    private String email;

}
```

So, what does this `register` endpoint actually do?

1. The `@POST` annotation lets us know that, in order to invoke this endpoint, we need to make an HTTP POST request
2. The `RegisterAccountRequest` argument to the `register` endpoint lets us know that we can submit a payload with our POST request,

which JAX-RS will then try to transform into an instance of this `RegisterAccountRequest` POJO

3. The `@Consumes(MediaType.APPLICATION_JSON)` annotation lets JAX-RS know that incoming request payloads are expected to be of type JSON (as opposed to, say, XML), which means that clients of this API should make sure that their JSON payloads can be transformed into instances of the `RegisterAccountRequest` POJO

After redeploying these changes to WildFly, we can test the new POST endpoint via the following curl request.

```
curl -iX POST http://localhost:8080/store/api/accounts \
  -H 'Content-Type: application/json' \
  -d '{ "name": "jessy", "email": "jessy@example.com" }'
```

It should yield the following HTTP response.

```
HTTP/1.1 200 OK

{"id":1,"name":"jessy","email":"jessy@example.com"}
```

(It's important to explicitly set the `Content-Type` request header to `application/json` when invoking the endpoint, or it won't work.)

Bean Validation

The `RegisterAccountRequest` POJO from the previous section is missing some *validation*. At the moment, we're blindly accepting any and all input in the `register` endpoint without checking, for example, whether a valid e-mail address was submitted: `abcdef@@@`, `!@#*$(@#$` and `1..2..3` are all accepted as e-mail addresses in our current implementation.

That's why Java EE offers us the **Bean Validation API** (JSR 380) for basic validation. It allows us to use annotations from the `javax.validation` package, like `@Max(30)`, `@NotNull`, `@NotBlank` and even `@Email`, and add these to the fields of our request POJOs.

```
package com.jessym.store.resources.dto;

import lombok.Data;

import javax.validation.constraints.Email;
import javax.validation.constraints.NotBlank;

@Data
public class RegisterAccountRequest {

    @NotBlank
    private String name;

    @Email
    @NotBlank
    private String email;

}
```

The last step, is to add the `@javax.validation.Valid` annotation to our `register` endpoint request parameter, which will trigger the actual validation for all incoming HTTP requests.

```
import javax.validation.Valid;

    // ...

    @POST
    public Account register(@Valid RegisterAccountRequest request) {
        Account account = new Account();
        account.setId(1L);
        account.setName(request.getName());
        account.setEmail(request.getEmail());
        return account;
    }

    // ...
```

After deploying these changes to WildFly, you'll find that you are greeted with a *400 Bad Request* HTTP response whenever you try to submit an invalid e-mail address or a blank name.

For example, the curl request below

```
curl -iX POST http://localhost:8080/store/api/accounts \
  -H 'Content-Type: application/json' \
  -d '{ "name": " ", "email": "!@#" }'
```

should now yield the following HTTP error response:

```
HTTP/1.1 400 Bad Request
```

Note that annotations like `@NotNull`, `@Min` and `@Max` can be applied to both number fields and string fields, whereas annotations like `@Email` and `@Pattern` (for regex pattern matching) can only be applied to string fields.

For an overview of all validation annotations, make sure to check out the **javax.validation.constraints** package.

Beans, Scopes and Injection

Let's go back to our `PingResource` for a moment.

```
@Path("/ping")
public class PingResource {

    @GET
    public Response ping() {
        return Response.ok(hashCode()).build();
    }
}
```

You'll find that each HTTP request to this **/ping** endpoint results in a different number in the HTTP response. And based on the code sample above, this must mean that a new instance of the `PingResource` class is being created for every HTTP request—hence the new hash code every time.

So what's going on here?

In order to understand this, we'll have to dive into the world of CDI, beans and EJBs. My goal for this section would be for you to understand the following sentence by the end of it:

The `PingResource` used to be a request-scoped CDI bean, but we changed it to a singleton EJB.

Beans and Scopes

In the Java EE world, certain class instances can be *under management* by the application server. This means that the server automatically instantiates your classes, and discards them again after some time. The implication of this, is that you never use the Java `new` keyword for these managed classes, because the application server creates (and destroys) them for you.

Such managed class instances are referred to as **beans**.

In addition, every bean comes with a so-called *scope* which determines its **life cycle**, i.e.: when the application server should instantiate or destroy it. Take our `PingResource`, for example. We haven't explicitly declared any scope annotations, but, in this case, the class-level `@Path` annotation causes the application server to register instances of this class as *request-scoped* beans.

The **request scope** means that a new bean is instantiated for *each and every* HTTP request, which explains why the refreshing of **/ping** yields a different hash code each time.

To be explicit about the scope of our `PingResource`, let's just add the `@RequestScoped` annotation:

```java
import javax.enterprise.inject.RequestScoped;

@Path("/ping")
@RequestScoped
public class PingResource {
  // ...
}
```

After deploying these changes to WildFly, you'll find that the **/ping** endpoint still exhibits the exact same behaviour, where it returns a random hash code on every refresh.

There are other scopes too:

- **@RequestScoped** - a new bean instance is created for each and every HTTP request
- **@SessionScoped** - a new bean instance is created and shared among all HTTP requests which are part of the same session *(we're not going to use the session scope in this article, but just to give some explanation: Java EE offers an authentication mechanism where the application server keeps track of a session token which the browser sends along with every HTTP request in the form of a cookie - this is how the server can identify a "logged in" user across multiple HTTP requests, and this is also how the server can instantiate and re-use the same session-scoped bean instance for all HTTP requests presenting the same cookie)*

- **@ApplicationScoped** - a single bean instance is created for the entire application and shared among all HTTP requests

Since it's almost always better to keep your beans free of internal state (stateless), I will recommend only ever declaring application-scoped beans for now. At least until we've discussed singletons.

Given this recommendation, let's change our `PingResource` from `@RequestScoped` to `@ApplicationScoped`:

```
import javax.enterprise.inject.ApplicationScoped;

@Path("/ping")
@ApplicationScoped
public class PingResource {
  // ...
}
```

After redeploying this updated application to WildFly, you'll find that refreshing the **/ping** endpoint now yields the exact same hash code every time, correctly indicating that there's only a single `PingResource` in our entire application, shared among *all* HTTP requests.

Java beans are a pretty interesting concept in and of themselves, but it becomes even *more* interesting (and much more powerful) once we learn that a Java bean can *inject* other beans.

CDI

CDI is an acronym for **Contexts and Dependency Injection**, which is another one of those Java EE APIs (JSR 365). In short, it describes some kind of mechanism where beans can have dependencies on other beans.

For example, consider the following three application-scoped beans.

```
@Path("/resource")
@ApplicationScoped
public class MyResource {

  @GET
  public void endpoint() {
    System.out.println("My Resource");
  }

}
@ApplicationScoped
public class ServiceA {

  public void printA() {
    System.out.println("A");
  }

}
@ApplicationScoped
public class ServiceB {

  public void printB() {
    System.out.println("B");
  }

}
```

If our resource bean requires functionality from services A and B to carry out its tasks, it can *inject* them as follows:

```
import javax.inject.Inject;

@Path("/resource")
@ApplicationScoped
public class MyResource {

  @Inject
  ServiceB serviceA;

  @Inject
  ServiceC serviceB;

  @GET
  public void endpoint() {
    System.out.println("My Resource");
    serviceA.printA();
    serviceB.printB();
  }

}
```

An HTTP request to the `/resource` endpoint would yield the following output in the server logs:

```
My Resource
A
B
```

To take it one step further, service A could *also* have a dependency on service B:

```
@ApplicationScoped
public class ServiceA {

  @Inject
  ServiceB serviceB;

  public void printA() {
    System.out.println("A");
    serviceB.printB();
  }

}
```

In this case, a request to the `/resource` endpoint will yield the following output:

```
My Resource
A
B
B
```

Here's what the corresponding dependency diagram looks like:

It's always important to make sure that there aren't any cycles in your application's dependency diagram. Even though the Java EE application server *might* be able to start an appliation with cyclic dependencies, it's usually considered a serious anti-pattern if `MyResource` depends on `ServiceA`, which depends on `ServiceB`, which depends on `MyResource` again.

In a typical 3-tier architecture, consisting of a resource layer (http), a service layer and a repository layer (database), you shouldn't run into any problems with cyclic dependencies, because all dependency arrows point in the same direction.

And it would be much better of our own project would follow this basic architecture as well, because there's way too much logic inside our `AccountResource` bean at the moment.

So, let's:

- introduce an `AccountService` (inside a `services` package)
- introduce an `AccountRepository` (inside a `persistence` package)
- change our `AccountResource` to be application-scoped
- move all `AccountResource` logic to the service and repository layers

```
package com.jessym.store.resources;

@Path("/accounts")
@Produces(MediaType.APPLICATION_JSON)
@Consumes(MediaType.APPLICATION_JSON)
@ApplicationScoped
public class AccountResource {

    @Inject
    AccountService accountService;

    @POST
    public Account register(@Valid RegisterAccountRequest request) {
        return accountService.register(request.getEmail(),
request.getName());
```

```
    }

    @GET
    public List<Account> list() {
        return accountService.list();
    }

    @GET
    @Path("/{id}")
    public Account findById(@PathParam("id") Long id) {
        return accountService.findById(id);
    }

}
package com.jessym.store.services;

@ApplicationScoped
public class AccountService {

    @Inject
    AccountRepository accountRepository;

    public Account register(String name, String email) {
        return accountRepository.register(name, email);
    }

    public List<Account> list() {
        return accountRepository.list();
    }

    public Account findById(Long id) {
        return accountRepository.findById(id);
    }

}
package com.jessym.store.persistence;

@ApplicationScoped
public class AccountRepository {

    public Account register(String name, String email) {
        Account account = new Account();
        account.setId(1L);
        account.setName(name);
        account.setEmail(email);
        return account;
    }

    public List<Account> list() {
        Account account = new Account();
        account.setId(1L);
        account.setName("bob");
        account.setEmail("bob@example.com");
        return List.of(account);
    }

    public Account findById(Long id) {
        Account account = new Account();
        account.setId(id);
        account.setName("alice");
        account.setEmail("alice@example.com");
        return account;
    }

}
```

If you deploy these changes to WildFly, you'll find that the application hasn't changed on a functional level because every endpoint still returns the same piece of JSON in its HTTP response:

```
curl -iX POST http://localhost:8080/store/api/accounts \
  -H 'Content-Type: application/json' \
  -d '{ "name": "jessy", "email": "jessy@example.com" }'

HTTP/1.1 200 OK
{"id":1,"name":"jessy","email":"jessy@example.com"}
curl -iX GET http://localhost:8080/store/api/accounts

HTTP/1.1 200 OK
[{"id":1,"name":"bob","email":"bob@example.com"}]
curl -iX GET http://localhost:8080/store/api/accounts/42

HTTP/1.1 200 OK
{"id":42,"name":"alice","email":"alice@example.com"}
```

The main difference is that our application is better organized and more flexible with this layered architecture. We might even introduce an `EmailService` inside our `services` package, which the `AccountService` can use to send newly registered customers a welcome e-mail.

```
@Slf4j
@ApplicationScoped
public class EmailService {

    // I'm using Lombok's @Slf4j annotation to automatically generate
    // a "private static final org.slf4j.Logger log" field variable

    public void sendEmail(Account account) {
        log.info("Thanks for signing up {} ({})", account.getEmail(),
account.getName());
    }

}
```

We can then update our `AccountResource` to inject this new bean, and send an e-mail upon registration as follows:

```
@ApplicationScoped
public class AccountService {

    @Inject
    AccountRepository accountRepository;

    @Inject
    EmailService emailService;

    public Account register(String name, String email) {
        Account account = accountRepository.register(name, email);
        emailService.sendEmail(account);
        return account;
    }

    // ...

}
```

After a redeployment, invoking the `/accounts` endpoint with a POST request should yield the following output in the server logs.

```
INFO [...] (...) Thanks for signing up jessy (jessy@example.com)
```

With the `EmailService` in place, our application finally matches the **dependency diagram** presented near the beginning of this article.

EJBs

The previously discussed scopes (*request*, *session* and *application*) are all specific to CDI. As such, any bean with one of these scopes is referred to as a **CDI bean**. In our case, this makes our application's `PingResource`, `AccountResource`, `AccountService`, `AccountRepository` and `EmailService` all examples of CDI beans.

But there's another type of bean: the **Enterprise Java Bean**, or **EJB**.

The biggest similarity between EJBs and CDI beans, is that EJBs can also *inject* other EJBs.

One of the differences between CDI beans and EJBs, is how they're *defined*. EJBs use a slightly different set of (EJB) scopes, such as:

- **@Stateless** - these beans are supposed to be free of internal state, so the application server can set up a *pool* of bean instances, without any guarantee about which particular bean instance will be invoked during a request
- **@Singleton** - a single bean instance is created for the entire application and shared among everyone and everything

---

⚠️ Warning ⚠️

In this article, `@Singleton` *always* refers to the `@javax.ejb.Singleton` annotation, and *never* to the `@javax.inject.Singleton` annotation.

Note how the EJB `@Singleton` scope is pretty similar to the CDI `@ApplicationSoped` annotation. That's because there's another, much more important difference between EJBs and CDI beans: *automatic container-managed transaction demarcation*.

This is a concept which sounds complicated, and *is* pretty complicated as well .A very basic summary would be that all of an EJBs public methods are executed in a transactional context, meaning that the container (application server) makes sure that either *all* of the method's operations succeed, or *none* of them (where it will *roll back* certain operations if necessary). Later on, we'll be discussing container-managed transactions in greater depth.

For now, I simply recommend changing all of the application's CDI beans to EJBs by changing their scope from `@ApplicationScoped` to `@javax.ejb.Singleton`.

And hopefully, this section has already given you a slightly better understanding of the sentence I posted earlier.

The `PingResource` used to be a request-scoped CDI bean, but we changed it to a singleton EJB.

Containers

Whenever discussing Java EE application servers, it's very likely for the term **container** to come up as well. That's because the container is a very important component of the Java EE application server: so important that the terms "application server" and "container" are often used interchangeably.

In short, the container is a Java EE server component, responsible for managing the lifecycle and injection of Java beans.

For example, if your application-scoped `AccountResource` bean attempts to inject an application-scoped `AccountService` bean, then it's the container which:

- creates a single instance of each class, and keeps track of these two instances forever (because of their scope)
- sets the `AccountService` instance as a field property on the `AccountResource` instance (at the position of the `@Inject` annotation)

And similarly, for request-scoped beans, it would also be the container which creates a *new* class instance for every HTTP request.

So, in summary, the container is tasked with figuring out the dependency diagram and scope overview of all Java beans in your application, and wiring everything together.

There's just one more important distinction to be made. A full Java EE application server actually comes equipped with *two* containers: the **servlet (or web) container** and the **EJB container**, responsible for managing CDI beans and EJBs respectively.

The EJB container is more advanced because, in addition to EJB lifecycle management and injection, it also provides this feature of automatic transaction demarcation for all its EJBs (more on this later).

This also ties back to the Java EE *web server* vs. *application server* distinction from earlier:

- A Java EE *application server* like WildFly or WebSphere, which complies with the full Java EE specification, comes with both a servlet container **and** an EJB container
- A Java EE *web server* like Tomcat or Jetty, which only complies with the web layer of the Java EE specification, **only** comes with a servlet container

Database Connectivity

Most Java EE applications have to connect to a database at some point. And for most corporate environments, somebody high up in the organization chooses which database must be used, depending on various multiyear and multimillion dollar support contracts which have already been signed. This choice typically lands on Oracle Database, IBM DB2 or Microsoft SQL Server.

For the build-along application of this article, I've decided to go with PostgreSQL instead. Arguably, one of the most advanced, enterprise grade, battle-hardened databases out there. Plus, it's 100% free for both personal and commercial use. Keep in mind, though, that Java EE + PostgreSQL is a combination you're unlikely to come across in corporate environments.

Because PostgreSQL is a bit different from other databases, I've included a small overview of the terminology below:

- a single PostgreSQL **instance** can host multiple **databases**
- a single **database** can have many different **tables**
- each **table** is prefixed with a certain **schema**, which can be seen as some kind of extra namespace within the database

Local PostgreSQL Setup via Docker

PostgreSQL can be downloaded and installed via their official website, but if you've already got Docker Compose installed on your system, there's a much easier way.

Create a `docker-compose.yaml` file at the root of your project) and run the `docker-compose up` command inside a terminal window in the same directory. This will automatically:

- pull the specified PostgreSQL Docker image
- set up a new database
- create a new user with full access to this database
- expose the database on the standard PostgreSQL port 5432

We're good to go, once the following log message is printed in the terminal window where `docker-compose up` is running:

```
database system is ready to accept connections
```

To shut this local PostgreSQL instance down, press Cmd + C or Ctrl + C in the terminal window. To bring it back up, just re-run the `docker-compose up` command, which will start the previously created PostgreSQL instance again.

While we're at it, let's also go ahead and manually run the DDL statements for setting up our application's database schema. Later on, we'll look into a better way of doing this automatically during deployment.

Make sure the `java_ee_postgres` container is running, which can be checked via the `docker ps` command, and execute the following command to start a PostgreSQL CLI session inside the container:

```
docker exec -it java_ee_postgres \
  bash -c "PGPASSWORD='password'; psql 'user=admin dbname=postgres'"
```

To create a database table called `account` in our database's `public` schema, run the following script.

```
CREATE TABLE account
(
    id    SERIAL PRIMARY KEY,
    name  VARCHAR,
    email VARCHAR UNIQUE
);
```

(In PostgreSQL, `SERIAL` is an auto-incrementing number field.)

The new table can then be queried in the same CLI session, via its fully qualified name.

```
SELECT * FROM public.account;
```

To end the PostgreSQL CLI session, simply run the `\q` command.

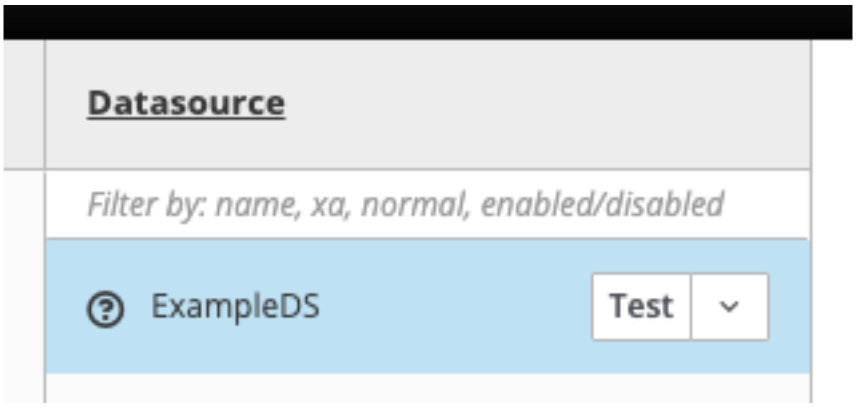
Registering PostgreSQL as a Data Source

In Java EE, *data sources* are the objects which represent the actual connection between an application and a particular database.

What's interesting, is that WildFly actually comes with a pre-configured data source for the **H2** in-memory database. If you open up the WildFly management console at **localhost:9990**, log in with the credentials entered earlier during the `./add-user.sh` script, and

1. click on **Runtime** (at the top),
2. click on **<computer name>** (in the **Server** column),
3. click on **Datasources** (in the **Monitor** column),

then you'll notice that there's already a data source called **ExampleDS**.



In order to add our own PostgresDS data source here, we first need to download the PostgreSQL JDBC driver from this **downloads page**. I recommend leaving the `postgresql-xx.jar` file inside your `~/Downloads/` folder for easy reference.

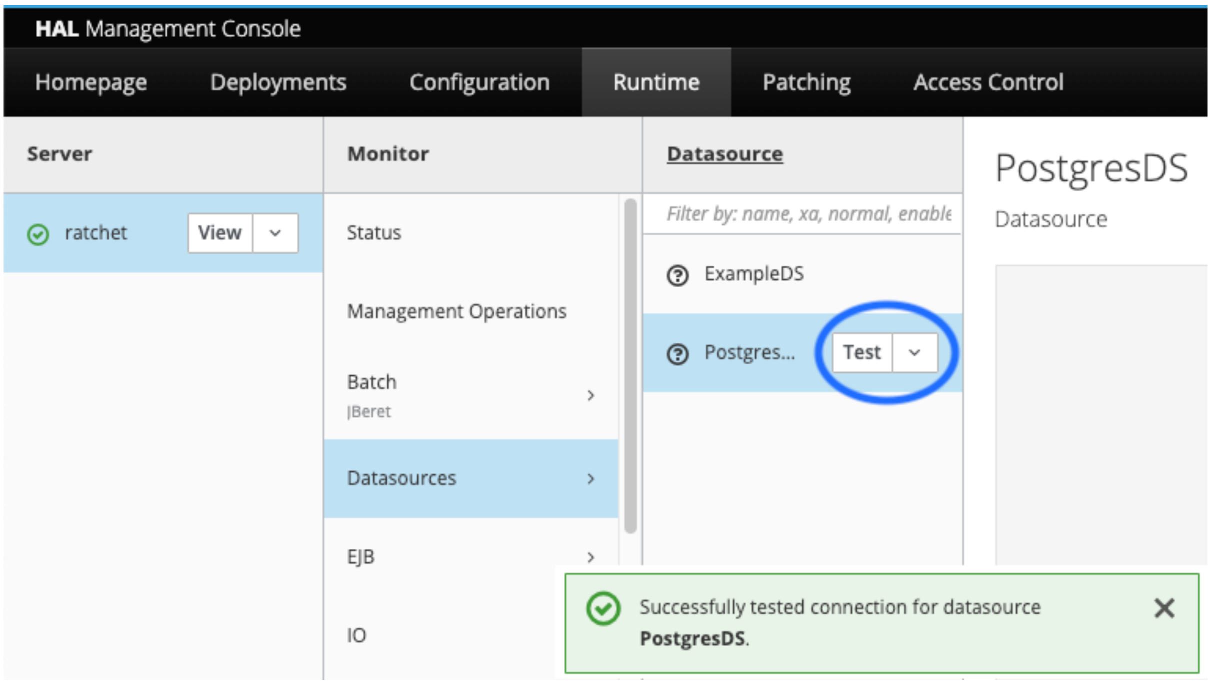
Next, open up a terminal window and navigate to the `$JBOSS_HOME/bin/` folder. After making sure that WildFly is running, execute the `./jboss-cli.sh -c` command to start a CLI session, and paste the following three commands to register PostgreSQL as a data source:

```
module add \
  --name=org.postgresql \
  --resources=~/Downloads/postgresql-42.2.16.jar \
  --dependencies=javax.api,javax.transaction.api
/subsystem=datasources/jdbc-driver=postgres:add( \
  driver-name="postgres", \
  driver-module-name="org.postgresql", \
  driver-class-name="org.postgresql.Driver" \
)
data-source add \
  --jndi-name=java:jboss/datasources/PostgresDS \
  --name=PostgresDS \
  --connection-url=jdbc:postgresql://localhost:5432/postgres \
  --driver-name=postgres \
  --user-name=admin \
  --password=password
```

Make sure to reference the correct driver (.jar) file in the first command, and the correct connection URL, username and password (from **docker-compose.yaml**) in the third command.

Also, make note of the data source's *JNDI* name, specified in the third command: `java:jboss/datasources/PostgresDS`. The JNDI name is a unique identifier which we're soon going to need in order to reference this new data source.

If the commands above have all completed without any errors, you can exit out of the CLI session by simply writing `exit`, or by pressing Cmd + C or Ctrl + C. To verify the connection, restart the WildFly application server (important!), navigate to the **Datasources** section WildFly's **management console**, and click the small **Test** button next to **PostgresDS**.



Successfully tested connection for datasource **PostgresDS**.

If the connection test fails, please make sure that a local PostgreSQL server is actually running on port 5432 with the right credentials. Also, if you ever need to make changes to the data source's configuration (connection URL, credentials, etc.), simply update the `standalone.xml` file inside `$JBOSS_HOME/standalone/configuration/` where the above settings are stored under the `<datasources>` XML element (don't forget to restart WildFly afterwards).

Persistence and Entities (JPA)

There are different ways to connect to a database from a Java EE application. The most basic approach would be to write a SQL query like `SELECT id, name, email FROM account;` yourself, execute it against the database, and iterate over the returned rows to populate the `id`, `name` and `email` fields of a list of `Account` POJOs.

Because these mappings between database rows and objects are such a common occurrence across all sorts of applications in all sorts of programming languages, a language-agnostic pattern called **ORM** (object-relational mapping) has emerged.

In Java EE, this ORM pattern is formalized by the **JPA** specification (JSR 338), which is an acronym for *Java Persistence API*. The idea is that, by adding just a few annotations like `@Table`, `@Id` and `@Column` to our `Account` POJO, the JPA implementation library will be able to figure out how this object needs to be mapped to the PostgreSQL `account` table we've set up earlier.

Let's start by adding some of these annotations to our `Account` POJO.

```java
package com.jessym.store.model;

import lombok.Data;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;

@Data
@Entity
@Table(name = "account")
public class Account {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "id")
    private Long id;

    @Column(name = "name")
    private String name;

    @Column(name = "email")
    private String email;

}
```

Here's an overview of what these annotations mean.

- **@Entity** - This lets JPA (or rather: the JPA implementation library) know that this is one of those database mapping objects which it should scan and index
- **@Table** - This lets JPA know the name of the database *table* to which this entity should be mapped
- **@Column** - This lets JPA know the name of the database *column* to which this particular field should be mapped
- **@Id** - This lets JPA know that a field corresponds to the database table's primary key column
- **@GeneratedValue** - Every relational database has this feature of auto-incrementing column values, so this annotation (together with the `strategy` parameter) lets JPA know that, even though there's a primary key constraint on the `id` column, we're still allowed to have our POJO's `id` field set to `null` when saving a new account (because the database will generate the `id` *for us*)

With these annotations in place, let's turn our attention to the `AccountRepository`, which will be responsible for saving and retrieving these `Account` objects from the database.

The first thing we need to do, is inject a so-called **entity manager** into this repository bean. This is some kind of utility object (provided by JPA) with methods like `persist` to actually save a new `Account` object into the database. We can inject and use the entity manager as follows:

```java
package com.jessym.store.persistence;

import com.jessym.store.model.Account;

import javax.ejb.Singleton;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;

@Singleton
public class AccountRepository {

    @PersistenceContext(unitName = "PostgresPU")
    EntityManager em;

    public Account register(String name, String email) {
        Account account = new Account();
        account.setName(name);
        account.setEmail(email);
        em.persist(account);
        return account;
    }

    // ...

}
```

In a nutshell, this repository:

1. is marked with the `@javax.ejb.Singleton` scope, because it should be an EJB for our purposes
2. attempts to inject an entity manager via some *persistence unit* named `PostgresPU` (more on this later)
3. uses the `em::persist` method to save a new account object into the database (where it omits the `id` field, because that will be auto-generated as declared by our table definition from earlier)
4. returns the `Account` entity, at which point the account would have been saved as a new row in the database, and at which point JPA would have set the `Account` POJO's `id` field *for us*, so it's no longer `null`

We can use the injected entity manager to connect the two remaining methods of `AccountRepository` (`list` and `findById`) to the database as well. But because JPA's criteria API (used for querying) can be a bit difficult to work with.

We're *almost* ready to connect to PostgreSQL from our Java application, because we still need to set up a **persistence unit**. Attempting to deploy all of the discussed changes to WildFly is likely to yield the following error:

```
WFLYJPA0033: Can't find a persistence unit named PostgresPU in
deployment
```

In order to resolve this final issue, we need to introduce one more deployment descriptor. We already have **web.xml** and **jboss-web.xml** inside `src/main/webbapp/WEB-INF/`, but we also need to add a file called **persistence.xml** inside `src/main/resources/META-INF/` with the following content:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence
        xmlns="http://java.sun.com/xml/ns/persistence"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd"
        version="1.0"
>

    <persistence-unit name="PostgresPU" transaction-type="JTA">
        <jta-data-source>java:jboss/datasources/PostgresDS</jta-data-
source>
        <properties>
            <property name="hibernate.dialect"
value="org.hibernate.dialect.PostgreSQL95Dialect" />
            <property name="hibernate.default_schema" value="public" />
        </properties>
    </persistence-unit>

</persistence>
```

Most important of all, we're actually defining a `PostgresPU` persistence unit here, which is already being referenced by our application's `AccountRepository`.

Second, we're referencing the `PostgresDS` data source via its JNDI name (which we've set up earlier), allowing us to link it to this new `PostgresPU` persistence unit.

And third, we're adding some *Hibernate-specific* properties here.

*Remember how JPA is just a specification, consisting of interfaces and annotations. The application server provides the actual runtime JPA implementation library. And in case of WildFly, the chosen JPA implementation library is called **Hibernate**.*

By setting the vendor-specific `hibernate.default_schema` property to `public`, for example, we can reference our PostgreSQL `account` table from the Java `Account` entity as `@Table(name = "account")`, rather than `@Table(name = "public.account")`.

Keep in mind, though, that any Hibernate-specific properties would no longer be picked up if we would ever swap our JPA implementation library from Hibernate to EclipseLink (something which would never happen for a sizeable application in the real world).

And that should be it! 😁

Make sure that PostgreSQL is running, and redeploy the latest version of the application to Wildfy. The `AccountResource` endpoints below should now be fully functional and result in the creation and querying of rows in PostgreSQL.

```
curl -iX POST http://localhost:8080/store/api/accounts \
  -H 'Content-Type: application/json' \
  -d '{ "name": "jessy", "email": "jessy@example.com" }'
curl -iX GET http://localhost:8080/store/api/accounts
curl -iX GET http://localhost:8080/store/api/accounts/1
```

After saving some new accounts into the database via the above registration endpoint, they should also be queryable directly via the PostgreSQL CLI client:

```
docker exec -it java_ee_postgres \
  bash -c "PGPASSWORD='password'; psql 'user=admin dbname=postgres'"
```

```
SELECT * FROM public.account;
```

Flyway Database Migrations

We began this section about database connectivity with the setup of a local PostgreSQL instance, where we manually executed a DDL script against the database for setting up the `account` table.

Because this manual procedure isn't ideal, there's a great tool called **Flyway** which can be used to automatically execute database migration files against the database during startup. And because these database migration files are placed inside a folder within `src/main/resources/`, the database schema would be part of the same version-controlled source tree as the Java EE application itself.

To get started, add a compile dependency on Flyway to the application's **pom.xml** file:

```
<dependency>
    <groupId>org.flywaydb</groupId>
    <artifactId>flyway-core</artifactId>
    <version>6.5.5</version>
</dependency>
```

Next, we're going to create an EJB with a hook for the *application-startup* lifecycle event. Add a class called **FlywayMigrationExecutor** with the following content:

```
package com.jessym.store.persistence.flyway;

import org.flywaydb.core.Flyway;

import javax.annotation.PostConstruct;
import javax.annotation.Resource;
import javax.ejb.Singleton;
import javax.ejb.Startup;
import javax.ejb.TransactionManagement;
import javax.ejb.TransactionManagementType;
import javax.sql.DataSource;

@Startup
@Singleton
@TransactionManagement(TransactionManagementType.BEAN)
public class FlywayMigrationExecutor {

    @Resource(lookup = "java:jboss/datasources/PostgresDS")
    DataSource dataSource;

    @PostConstruct
    public void migrate() {
        Flyway flyway = Flyway.configure()
            .dataSource(dataSource)
            .schemas("public")
            .load();
        flyway.migrate();
    }

}
```

Here's a breakdown of all the annotations we're seeing here.

- The **@Startup** annotation declares that this bean should be initialized *eagerly* (i.e.: during the application's startup sequence)
- The **@Singleton** annotation declares this class to be an EJB
- The **@TransactionManagement** annotation lets WildFly know that container transaction management should be disabled for this EJB and that it will manage its own transactions (at the **BEAN** level) - this is necessary for Flyway because it needs to execute SQL queries which wouldn't normally be allowed in a container-managed environment
- The **@Resource** annotation lets us perform a JNDI lookup for the `PostgresDS` data source we created earlier
- The **@PostConstruct** annotation instructs WildFly to execute the `migrate` method once the container has successfully injected all instance fields (so in this case, we can be sure that the `dataSource` field is no longer `null` by the time `migrate` is invoked)

Apart from these annotations, we're using the Flyway API to initialize a new `Flyway` object, where we're required to pass in a `DataSource` object, and where we can optionally pass in our default PostgreSQL schema name.

Calling `flyway.migrate()` will actually trigger Flyway to look for database migration files inside the `src/main/resources/db/migration/` folder, and execute these against the provided data source object. So, taking into account Flyway's **naming conventions** for SQL migration files, let's add our first **V1__create_account_table.sql** migration file to this `src/main/resources/db/migration/` folder:

```
CREATE TABLE account
(
    id    SERIAL PRIMARY KEY,
    name  VARCHAR,
    email VARCHAR UNIQUE
);
```

After deploying these changes to WildFly for the first time, you're likely to be greeted by the following error message:

```
org.flywaydb.core.api.FlywayException:
  Found non-empty schema(s) \"public\" but no schema history table.
  Use baseline() or set baselineOnMigrate to true to initialize the
schema history table.
```

This basically means that WildFly has noticed that there's already an `account` table from earlier. The easiest way to deal with this, is by deleting the current PostgreSQL container via the `docker rm -f java_ee_postgres` command and re-creating it via `docker-compose up`.

Either redeploying the application or restarting the entire WildFly server should then yield the following log output from Flyway.

```
INFO [...] (...) Creating Schema History table
"public"."flyway_schema_history" ...
INFO [...] (...) Current version of schema "public": << Empty Schema >>
INFO [...] (...) Migrating schema "public" to version 1 - create
account table
INFO [...] (...) Successfully applied 1 migration to schema "public"
```

This means that, from now on, all database tables within the PostgreSQL `public` schema will be under management by Flyway. To this end, Flyway has automatically created a `flyway_schema_history` table which it uses to keep track of which database migration files have already been executed against this database, so it never runs the same migrations twice.

If we wanted to add an extra `tos_accepted` boolean column to the `account` table, then all we have to do is add a new database migration file to the same resources folder and redeploy the application to WildFly.

```sql
# src/main/resources/db/migration/V2__add_tos_accepted_colunm.sql
ALTER TABLE account ADD COLUMN tos_accepted BOOLEAN;
```

Keep in mind, though, that these files aren't supposed to be edited after they've been applied to the database. The Flyway SQL migration files should be thought of as an *append-only* collection, where the best approach is to *roll-forward* with a corrective migration if there's ever a typo or logical error in one of your migration files.

Check out the official **Flyway docs** for more information.

Container Transaction Management

One important difference between EJBs and ordinary Java beans, is that an EJB's public methods are always executed in a **transactional context**, where the EJB container makes sure that either *all* of the method's operations succeed, or *none* of them (where it will *roll back* if necessary).

Theoretically, this should make it easier to reason about the system, because (again: theoretically) it can never end up in a partial state where one operation has successfully completed, but another operation has not.

Assume we have an EJB like the one below.

```
@Singleton
public class EJB {

    public void someMethod() {
        // Persist a new entity into the database via JPA
        entityManager.persist(new Account("Jessy",
"jessy@example.com"));
        // Send a message to some queue via JMS
        jmsProducer.send(queue, "Hello World");
    }

}
```

An extreme oversimplification of how the EJB container actually invokes `someMethod()` would be something like this.

```
try {
    ejb.someMethod();
    commit();
} catch (Exception e) {
    rollback();
    throw e;
}
```

So, what does `commit()` and `rollback()` even mean?

Well, that's the part where it starts getting interesting. In the example above, our `someMethod()` includes a persistence operation towards some relational database via JPA, followed by the sending of some message to a queue via JMS (out of scope for this article).

Both of these external components (the relational database and the JMS broker) already support the concept of transactional operations in and of themselves:

- it's possible to already send a bunch of SQL statements to a **relational database**, but wait some time before actually *committing* these statements (and executing them for real)
- it's possible to already prepare a message for some **JMS broker's queue**, but wait some time before actually *committing* the message (and sending it for real)

These already-existing mechanisms allow the EJB container to simply hook into them, and provide this *commit-or-rollback* feature across multiple operations.

For example, if our EJB's `someMethod()` were to look like this:

```java
public void someMethod() {
    entityManager.persist(new Account("Jessy", "jessy@example.com"));
    if (1 + 1 == 2) {
        throw new RuntimeException();
    }
    jmsProducer.send(queue, "Hello World");
}
```

Then the EJB container won't ever *commit* the `entityManager`'s database operation when the runtime exception is thrown, effectively rolling it back.

And it's important to reiterate that the EJB container can only offer this commit-or-rollback feature for operations against external systems which *already* support this. Like relational databases and JMS brokers.

If our EJB's `someMethod()` were to look like this:

```java
public void someMethod() {
    entityManager.persist(new Account("Jessy", "jessy@example.com"));
    httpClient.send(httpRequest, bodyHandler);
    jmsProducer.send(queue, "Hello World");
}
```

Then, if the final JMS operation would fail, the EJB container **wouldn't** be able to rollback the outbound HTTP request, because who knows what has already happened on the receiving end of the HTTP request. The EJB container *would* still rollback the database operation, though.

The execution of a bean's public methods in a transactional context is enabled by default for all EJBs. In order to **disable** this feature, the EJB should be decorated with the following class-level annotation.

```java
import javax.ejb.TransactionManagement;
import javax.ejb.TransactionManagementType;

@TransactionManagement(TransactionManagementType.BEAN)
```

In order to **enable** this feature for ordinary (CDI) Java beans, they should be decorated with the `@javax.transaction.Transactional` annotation at the method or class level.

Conclusion

The Java EE framework, with its steep learning curve, is difficult to get started with for newcomers. Not because something's inherently wrong with the technology, but because there's hardly any documentation