Bootique Documentation

Getting Started

Hello World in Bootique

The goal of this chapter is to write a simple REST app using Bootique. Let’s start with a new Java Maven project created in your favorite IDE. Your `pom.xml` in addition to the required project information tags will need to declare a few BOM ("Bill of Material") imports in the `<dependencyManagement/>` section:

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>io.bootique.bom</groupId>
      <artifactId>bootique-bom</artifactId>
      <version>0.25</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

This will allow `<dependencies/>` section that will follow shortly to include various Bootique modules and not worry about their individual versions. So let’s create this section and import a few modules:

```
<dependencies>
  <dependency>
    <groupId>io.bootique.jersey</groupId>
    <artifactId>bootique-jersey</artifactId>
    <scope>compile</scope>
  </dependency>
  <dependency>
    <groupId>io.bootique.logback</groupId>
    <artifactId>bootique-logback</artifactId>
    <scope>compile</scope>
  </dependency>
</dependencies>
```

As you see we want a `bootique-jersey` and `bootique-logback` modules in our app. Those may depend on other modules, but we don’t have to think about it. Those dependencies will be included by Maven automatically. Now let’s create the main Java class that will run the app:

```
package com.foo;

import io.bootique.Bootique;

public class Application {

    public static void main(String[] args) {
        Bootique
            .app(args)
            .autoLoadModules()
            .exec()
            .exit();
    }
}
```

There’s only one line of meaningful code inside the `main()` method, but this is already a working Bootique app. Meaning it is runnable and can do a few things. So let’s try running this class from your IDE. You will see the output like this on the IDE console:

```
NAME
      com.foo.Application

OPTIONS
      -c yaml_location, --config=yaml_location
           Specifies YAML config location, which can be a file path or a URL.

      -h, --help
           Prints this message.

      -H, --help-config
           Prints information about application modules and their configuration
           options.

      -s, --server
           Starts Jetty server.
```

So the app printed its help message telling us which commands and options it understands. `--server` option looks promising, but before we use it, let’s actually write a REST endpoint that will answer to our requests. We’ll use standard Java JAX-RS API for that:

```
package com.foo;

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

@Path("/")
public class HelloResource {

    @GET
    public String hello() {
        return "Hello, world!";
    }
}
```

Note that we could have placed this code straight in the Main class. Which makes for an effective demo (an app that can fit in one class), but not for a particularly clean design. So keeping the resource in its own class. Now let’s amend the `Main` class to tell Bootique where to find the resource:

```
package com.foo;

import com.google.inject.Module;
import io.bootique.Bootique;
import io.bootique.jersey.JerseyModule;

public class Application {

    public static void main(String[] args) {

        Module module = binder ->
                  JerseyModule.extend(binder).addResource(HelloResource.class);

        Bootique
            .app(args)
            .module(module)
            .autoLoadModules()
            .exec()
            .exit();
    }
}
```

Here we created our own module that "contributes" resource configuration to the JerseyModule. Now let’s try to run the app with the changes. Add `--server` to the command run parameters before doing it. Now when the app is started, you will see different output:

```
INFO main o.e.jetty.util.log: Logging initialized @1328ms
INFO main i.b.j.s.ServerFactory: Adding listener
io.bootique.jetty.servlet.DefaultServletEnvironment
INFO main i.b.j.s.ServletFactory: Adding servlet 'jersey' mapped to /*
INFO main i.b.j.s.ServerLifecycleLogger: Starting jetty...
INFO main o.e.j.server.Server: jetty-9.3.6.v20151106
INFO main o.e.j.s.h.ContextHandler: Started
o.e.j.s.ServletContextHandler@27dc79f7{/,null,AVAILABLE}
INFO main o.e.j.s.ServerConnector: Started
ServerConnector@3a45c42a{HTTP/1.1,[http/1.1]}{0.0.0.0:8080}
INFO main o.e.j.server.Server: Started @2005ms
INFO main i.b.j.s.ServerLifecycleLogger: Started Jetty in 584 ms. Base URL:
http://127.0.0.1:8080/
```

Notice that the app did not terminate immediately, and is waiting for user requests. Now let’s try opening the URL http://localhost:8080/ in the browser. We should see 'Hello, world!' as request output. We just built a working REST app that does not require deployment to a web container, and generally wasn’t that hard to write. The takeaway here is this:

- You start the app via `Bootique` class, that gives you a runnable "shell" of your future app in one line of code.
- Declaring modules in the app dependencies and using `Bootique.autoLoadModules()` gives the app the ability to respond to commands from those modules (in our example `--server` command coming from implicit bootique-jetty module started an embedded web server).
- You can contribute your own code to modules to build an app with desired behavior.

Next we’ll talk about configuration…

Configuration

You can optionally pass a configuration to almost any Bootique app. This is done with a `--config` parameter. An argument to `--config` is either a path to a configuration file or a URL of a service that serves such configuration remotely (imagine an app starting on a cloud that downloads its configuration from a central server). The format of the file is YAML (though, just like everything in Bootique, this can be customized). Let’s create a config file that changes Jetty listen port and the app context path. To do this create a file in the app run directory, with an arbitrary name, e.g. `myconfig.yml` with the following contents:

```
jetty:
  context: /hello
  connectors:
    - port: 10001
```

Now restart the app with the new set of parameters: `--server --config=myconfig.yml`. After the restart the app would no longer respond at http://localhost:8080/, instead you will need to use a new URL: http://localhost:10001/hello. This is just a taste of what can be done with configuration. Your app can just as easily obtain its own specific configuration in a form of an app-specific object, as described elsewhere in the docs.

Injection

We’ve mentioned that Bootique is built on Google Guice dependency injection (DI) container. We’ll talk more about injection elsewhere. Here we’ll provide a simple example. Our simple app already has a number of objects and services working behind the scenes that can be injected. One of them is command-line arguments that were provded to the app on startup. Let’s extend our resource to include those arguments in the output:

```
package com.foo;

import static java.util.stream.Collectors.joining;

import java.util.Arrays;

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

import com.google.inject.Inject;
import io.bootique.annotation.Args;

@Path("/")
public class HelloResource {

    @Inject
    @Args
    private String[] args;

    @GET
    public String hello() {
        String allArgs = Arrays.asList(args).stream().collect(joining(" "));
        return "Hello, world! The app was started with the following arguments: " +
allArgs;
    }
}
```

As you see, we declared a variable of type `String[]` and annotated it with `@Inject` and `@Args`. `@Inject` (must be a `com.google.inject.Inject`, not `javax.inject.Inject`) ensures that the value is initialized via injection, and `@Args` tells Bootique which one of possibly many String[] instances from the DI container we are expecting here.

Now you can restart the app and refresh http://localhost:10001/hello in the browser. The new output will be "Hello, world! The app was started with the following arguments: `--server --config=myconfig.yml`".

Next let’s discuss how to build and run the app outside the IDE…

Packaging

Till now we've been running our app from IDE (which also happened to be much easier then running typical container-aware apps). Now let's package our app as a runnable "fat" jar to be able to run it from command line (e.g. in deployment environment). Assembling "fat" jar requires a bit of configuration of the Maven `maven-shade-plugin`. To simplify it, you can set a parent of your `pom.xml` to be a standard Bootique parent:

```
<parent>
    <groupId>io.bootique.parent</groupId>
    <artifactId>bootique-parent</artifactId>
    <version>0.12</version>
</parent>
```

Other required `pom.xml` additions:

```
<properties>
    <main.class>com.foo.Application</main.class>
</properties>
<!--...-->
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-shade-plugin</artifactId>
        </plugin>
    </plugins>
</build>
```

Once this is setup you can build and run the app:

```
mvn clean package

# Using myapp-1.0.jar as an example; the actual jar name depends on your POM settings
java -jar target/myapp-1.0.jar --server --config=myconfig.yml
```

The result should be the same as running from the IDE and the app should be still accessible at http://localhost:10001/hello. Now your jar can be deployed in any environment that has Java 8.

This concludes our simple tutorial. Now you can explore our documentation to read more about Bootique core and individual modules.

Bootique Core Documentation

Part I. Overview

Chapter 1. What is Bootique

Bootique is a minimally opinionated technology for building container-less runnable Java applications. No JavaEE container required to run your app! It is an ideal platform for *microservices*, as it allows you to create a fully functional app with minimal-to-no setup. Though it is not limited to a specific kind of app (or the "micro" size) and can be used for REST services, webapps, runnable jobs, DB migrations, JavaFX GUI apps to mention a few examples.

Unlike traditional container-based apps, Bootique allows you to control your `main()` method and create Java apps that behave like simple executable commands that can be run with Java:

```
java -jar my.jar [arguments]
```

Each Bootique app can be started with a YAML configuration loaded from a file or from a remote URL. Among other benefits, such configuration approach ideally suits cloud deployment environments.

Bootique was inspired by two similar products - Dropwizard and SpringBoot, however its focus is different. Bootique favors modularity and clean pluggable architecture. Bootique is built on top of Google Guice dependency injection (DI) container, which provides the core of its modularity mechanism. This means that pretty much anything in Bootique can be customized/overridden to your liking.

Chapter 2. Java Version

Java 8 or newer is required.

Chapter 3. Build System

Bootique apps can be built using any Java build system (Ant, Maven, Gradle, etc). Examples in the documentation are based on Maven and `maven-shade-plugin`. While this is not strictly a requirement, Bootique apps are usually packaged into "fat" runnable jars and don't have any external dependencies beyond the JRE.

Chapter 4. Programming Skills

Everything you know about Java programming will be applicable when working with Bootique. You may need to "unlearn" some of the practices related to JavaEE configuration and container deployment though.

Integration between various parts of a Bootique app is done via Google Guice. In most cases Bootique API would steer you towards idiomatic approach to integration, so deep knowledge of Guice is not required. Though it wouldn't hurt to understand a few main concepts: modules, bindings, multibindings, overrides.

Java ServiceLoader facility is another important part of Bootique, and probably yet another thing that you shouldn't worry too much about initially.

Part II. Programming

Chapter 5. Modules

Bootique apps are made of "modules". The framework simply locates all available modules, loads them in the DI environment, parses the command line, and then transfers control to a Command (that can originate from any of the modules) that matched the user choice. There's a growing list of modules created by Bootique development team. And you can easily write your own. In fact, programming in Bootique is primarily about writing Modules.

A module is a Java library that contains some code. What makes it a module is a special Java class that implements Guice Module interface. This class defines what "services" or other types of objects the module provides (in other words what will be injectable by the module users). This is done in a form of "bindings", i.e. associations between publicly visible injectable service interfaces and specific implementations:

```
public class MyModule implements Module {
    @Override
    public void configure(Binder binder) {
        binder.bind(MyService.class).to(MyServiceImpl.class);
    }
}
```

There are other flavors of bindings in Guice. Please refer to Guice documentation for details. One important form extensively used in Bootique is Multibinding.

Chapter 6. Modules Auto-Loading

Modules can be automatically loaded via `Bootique.autoLoadModules()` as long as they are included in your aplication dependencies. Auto-loading depends on the Java ServiceLoader mechanism. To ensure your modules can be auto-loaded do two things. First implement `io.bootique.BQModuleProvider` interface specific to your module:

```
public class MyModuleProvider implements BQModuleProvider {
    @Override
    public Module module() {
        return new MyModule();
    }
}
```

After that create a file `META-INF/services/io.bootique.BQModuleProvider` with the only line being the name of your BQModuleProvider implementor. E.g.:

```
com.foo.MyModuleProvider
```

`BQModuleProvider` has two more methods that you can optionally implement to help Bootique to make sense of the module being loaded:

```
public class MyModuleProvider implements BQModuleProvider {
    // ...

    // provides human-readable name of the module
    @Override
    public String name() {
        return "CustomName";
    }

    // a collection of modules whose services are overridden by this module
    @Override
    public Collection<Class<? extends Module>> overrides() {
        return Collections.singleton(BQCoreModule.class);
    }
}
```

If in your Module you are planning to redefine any services from the upstream modules, specify those upstream modules in the `overrides()` collection. In practice overrides are rarely needed, and often can be replaced with service decomposition.

Chapter 7. Configuration and Configurable Factories

Bootique Modules obtain their configuration in a form of "factory objects". We'll show some examples shortly. For now let's focus on the big picture, namely the fact that Bootique app configuration is multi-layered and roughly follows the sequence of "code - config files (contributed) - config files (CLI) - overrides". "Code" is the default values that are provided in constructors of factory objects. Config files overlay those defaults with their own values. Config files can be either contributed in the code, or specified on the command line. Files is where the bulk of configuration usually stored. Finally config values may be further overridden via Java properties and/or environment variables.

Configuration via YAML Files

Format of configuration file can be either JSON or YAML. For simplicity we’ll focus on YAML format, but the two are interchnageable. Here is an example config file:

```
log:
  level: warn
  appenders:
    - type: file
      logFormat: '%c{20}: %m%n'
      file: target/logback/debug.log

jetty:
  context: /myapp
  connectors:
    - port: 12009
```

While not strictly required, as a rule the top-level keys in the file belong to configuration objects of individual modules. In the example above "log" subtree configures `bootique-logback` module, while "jetty" subtree configures `bootique-jetty`. For standard modules refer to module-specific documentation on the structure of the supported configuration (or run your app `-H` flag to print supported config to the console). Here we’ll discuss how to build your own configuration-aware module.

Bootique allows each Module to read its specific configuration subree as an object of the type defined in the Module. Very often such an object is written as a factory that contains a bunch of setters for configuration properties, and a factory method to produce some "service" that a Module is interested in. Here is an example factory:

```
public class MyFactory {

    private int intProperty;
    private String stringProperty;

    public void setIntProperty(int i) {
        this.intProperty = i;
    }

    public void setStringProperty(String s) {
        this.stringProperty = s;
    }

    // factory method
    public MyService createMyService(SomeOtherService soService) {
        return new MyServiceImpl(soService, intProperty, stringProperty);
    }
}
```

The factory contains configuration property declarations, as well as public setters for these properties (You can create getters as well. It is not strictly required, but may be useful for unit tests, etc.). Now let’s take a look at the Module class:

```
public class MyModule extends ConfigModule {
    @Provides
    public MyService createMyService(
             ConfigurationFactory configFactory,
             SomeOtherService soService) {

        return configFactory
                 .config(MyFactory.class, configPrefix)
                 .createMySerice(soService);
    }
}
```

And now a sample configuration that will work with our module:

```
my:
  intProperty: 55
  stringProperty: 'Hello, world!'
```

A few points to note here:

- Calling our module "MyModule" and extending from `ConfigModule` gives it access to the protected "configPrefix" instance variable that is initialized to the value of "my" (the naming convention here is to use the Module simple class name without the "Module" suffix and converted to lowercase).
- `@Provides` annotation is a Guice way of marking a Module method as a "provider" for a certain type of injectable service. All its parameters are themselves injectable objects.
- `ConfigurationFactory` is the class used to bind a subtree of the app YAML configuration to a given Java object (in our case - MyFactory). The structure of MyFactory is very simple here, but it can be as complex as needed, containing nested objects, arrays, maps, etc. Internally Bootique uses Jackson framework to bind YAML to a Java class, so all the features of Jackson can be used to craft configuration.

Configuration File Loading

There are a number of ways to pass a config file to a Bootique app, roughly falling in two categories - files contributed via DI and files passed on command line. Let’s discuss them one by one:

- Contributing a config file via DI:

  ```
  BQCoreModule.extend(binder).addConfig("classpath:com/foo/default.yml");
  ```

  A primary motivation for this style is to provide application default configuration, with YAML files often embedded in the app and read from the classpath (as suggested by the "classpath:.." URL in the example). More then one configuration can be contributed. E.g. individual modules might load their own defaults. Multiple configs are combined in a single config tree by the runtime. The order in which this combination happens is undefined, so make sure there are no conflicts between them. If there are, consider replacing multiple conflicting configs with a single config.

- Conditionally contributing a config file via DI. It is possible to make DI configuration inclusion conditional on the presence of a certain command line option:

  ```
  OptionMetadata o = OptionMetadata.builder("qa")
        .description("when present, uses QA config")
        .build();

  BQCoreModule.extend(binder)
        .addOption(o)
        .addConfigOnOption(o.getName(), "classpath:a/b/qa.yml");
  ```

- Specifiying a config file on command line. Each Bootique app supports `--config` option that takes a configuration file as its parameter. To specify more than one file, use `--config` option multiple times. Configurations will be loaded and merged together in the order of their appearance on the command line.
- Specifying a single config value via a custom option:

  ```
  OptionMetadata o = OptionMetadata.builder("db")
        .description("specifies database URL")
        .configPath("jdbc.mydb.url")
        .defaultValue("jdbc:mysql://127.0.0.1:3306/mydb")
        .build();

  BQCoreModule.extend(binder).addOption(o);
  ```

  This adds a new `--db` option to the app that can be used to set JDBC URL of a datasource called "mydb". If not specified, the default value provided in the code is used.

Configuration via Properties

YAML file can be thought of as a set of nested properties. E.g. the following config

```
my:
  prop1: val1
  prop2: val2
```

can be represented as two properties ("my.prop1", "my.prop2") being assigned some values. Bootique takes advantage of this structural equivalence and allows to define configuration via properties as an alternative (or more frequently - an addition) to YAML. If the same "key" is defined in both YAML file and a property, `ConfigurationFactory` would use the value of the property (in other words properties override YAML values).

To turn a given property into a configuration property, you need to prefix it with “bq.”. This "namespace" makes configuration explicit and helps to avoid random naming conflicts with properties otherwise present in the system.

Properties can be provided to Bootique via BQCoreModule extender:

```
class MyModule implements Module {
    public void configure(Binder binder) {

        BQCoreModule.extend(binder)
          .setProperty("bq.my.prop1", "valX");

        BQCoreModule.extend(binder)
                .setProperty("bq.my.prop2", "valY");
    }
}
```

Alternatively they can be loaded from system properties. E.g.:

```
java -Dbq.my.prop1=valX -Dbq.my.prop2=valY -jar myapp.jar
```

Though generally this approach is sneered upon, as the authors of Bootique are striving to make Java apps look minimally "weird" in deployment, and "-D" is one of those unintuitive "Java-only" things. Often a better alternative is to define the bulk of configuration in YAML, and pass values for a few environment-specific properties via shell variables (see the next section) or bind them to CLI flags.

Configuration via Environment Variables

Bootique allows to use *environment variables* to specify/override configuration values. While variables work similar to JVM properties, using them has advantages in certain situations:

- They may be used to configure credentials, as unlike YAML they won’t end up in version control, and unlike Java properties, they won’t be visible in the process list.
- They provide customized application environment without changing the launch script and are ideal for containerized and other virtual environments.
- They are more user-friendly and appear in the app help.

To declare variables associated with configuration values, use the following API (notice that no "bq." prefix is necessary here to identify the configuration value):

```
class MyModule implements Module {
    public void configure(Binder binder) {

        BQCoreModule.extend(binder)
               .declareVar("my.prop1", "P1");

        BQCoreModule.extend(binder)
               .declareVar("my.prop2", "P2");
    }
}
```

So now a person running the app may set the above configuration as

```
export P1=valX
export P2=valY
```

Moreover, explicitly declared vars will automatically appear in the application help, assisting the admins in configuring your app

(TODO: document BQConfig and BQConfigProperty config factory annotations required for the help generation to work)

```
$ java -jar myapp-1.0.jar --help
...
ENVIRONMENT
      P1
           Sets value of some property.

      P2
           Sets value of some other property.
```

NOTE Notice that previously used naming conventions to bind variables that start with `BQ_*` to config values are deprecated and support for them will be removed soon. Such approach was causing too much unexpected behavior in non-containerized environments. The alternative is explicitly declared variables described above.

Polymorphic Configuration Objects

A powerful feature of Jackson is the ability to dynamically create subclasses of the configuration objects. Bootique takes full advantage of this. E.g. imagine a logging module that needs "appenders" to output its log messages (file appender, console appender, syslog appender, etc.). The framework might not be aware of all possible appenders its users might come up with in the future. Yet it still wants to have the ability to instantiate any of them, based solely on the data coming from YAML. Moreover each appender will have its own set of incompatible configuration properties. In fact this is exactly the situation with `bootique-logback` module.

Here is how you ensure that such a polymorphic configuration is possible. Let's start with a simple class hierarchy and a factory that contains a variable of the supertype that we'd like to init to a concrete subclass in runtime:

```
public abstract class SuperType {
    // ...
}

public class ConcreteType1 extends SuperType {
    // ...
}

public class ConcreteType2 extends SuperType {
    // ...
}

public class MyFactory {

    // can be a class or an interface
    private SuperType subconfig;

    public void setSubconfig(SuperType s) {
        this.subconfig = s;
    }

    // ...
}
```

To make polymorphism work, we need to provide some instructions to Jackson. First we need to annotate the supertype and subtypes:

```
@JsonTypeInfo(use = JsonTypeInfo.Id.NAME,
     property = "type",
     defaultImpl = ConcreteType1.class)
public abstract class SuperType {

}

@JsonTypeName("type1")
public class ConcreteType1 extends SuperType {

}

@JsonTypeName("type2")
public class ConcreteType2 extends SuperType {

}
```

After that we need to create a service provider file called `META-INF/service/io.bootique.config.PolymorphicConfiguration` where all the types participating in the hierarchy are listed (including the supertype):

```
com.foo.SuperType
com.foo.ConcreteType1
com.foo.ConcreteType2
```

This should be enough to work with configuration like this:

```
my:
  subconfig:
    type: type2
    someVar: someVal
```

If another module decides to create yet another subclass of SuperType, it will need to create its own `META-INF/service/io.bootique.config.PolymorphicConfiguration` file that mentions the new subclass.

Chapter 8. Using Modules

Modules can use other "upstream" modules in a few ways:

- "Import": a downstream module uses another module as a library, ignoring its injectable services.
- "Use" : downstream module's classes inject classes from an upstream module.
- "Contribute": downstream module injects objects to collections and maps defined in upstream modules.

Import case is trivial, so we'll concentrate on the two remaining scenarios. We will use BQCoreModule as an example of an upstream module, as it is available in all apps.

Injecting Other Module's Services

You can inject any services declared in other modules. E.g. BQCoreModule provides a number of objects and services that can be accessed via injection:

```
class MyService {

    @Args
    @Inject
    private String[] args;

    public String getArgsString() {
        return Arrays.asList(getArgs()).stream().collect(joining(" "));
    }
}
```

In this example we injected command line arguments that were used to start the app. Note that since there can potentially be more than one `String[]` in a DI container, Bootique `@Args` annotation is used to uniquely identify the array that we want here.

Contributing to Other Modules

Guice supports multibindings, intended to *contribute* objects defined in a downstream module to collections/maps used by services in upstream modules. Bootique hides Guice API complexities, usually providing "extenders" in each module. E.g. the following code adds `MyCommand` the the app set of commands:

```
public class MyModule implements Module {

    @Override
    public void configure(Binder binder) {
        BQCoreModule.extend(binder).addCommand(MyCommand.class);
    }
}
```

Here we obtained an extender instance via a static method on BQCoreModule. Most standard modules define their own extenders accessible via "`extend(Binder)`". This is a pattern you might want to follow in your own modules.

Chapter 9. Application Class

A class that contains the "`main()`" method is informally called "application". Bootique does not impose any additional requirements on this class. You decide what to put in it. It can be limited to just "`main()`", or turned into a REST API resource, etc.

Application as a Module

Most often then not it makes sense to turn the application class into a Module though. After all a Bootique app is just a collection of Modules, and this way the application class would represent that one final Module to rule them all:

```
public class Application implements Module {

   public static void main(String[] args) {
      Bootique.app(args).module(Application.class).autoLoadModules().exec().exit();
   }

   public void configure(Binder binder) {
      // load app-specific services; redefine standard ones
   }
}
```

You may also implement a separate BQModuleProvider for the Application module. Then `autoLoadModules()` will discover it just like any other Module, and there won't be a need to add Application module explicitly.

Common Main Class

If all your code is packaged in auto-loadable modules (which is always a good idea), you may not even need a custom main class. `io.bootique.Bootique` class itself declares a `main()` method and can be used as an app launcher. This creates some interesting possibilities. E.g. you can create Java projects that have no code of their own and are simply collections of modules declared as compile dependencies. More details on packaging are given in the "Runnable Jar" chapter.

Chapter 10. Commands

Bootique runtime contains a set of commands coming from Bootique core and from all the modules currently in effect in the app. On startup Bootique attempts to map command-line arguments to a single command type. If no match is found, a *default* command is executed (which is normally a "help" command). To list all available commands, the app can be run with `--help` option (in most cases running without any options will have the same effect). E.g.:

```
$ java -jar myapp-1.0.jar --help

NAME
      com.foo.MyApp

OPTIONS
      -c yaml_location, --config=yaml_location
           Specifies YAML config location, which can be a file path or a URL.

      -h, --help
           Prints this message.

      -H, --help-config
           Prints information about application modules and their configuration
           options.

      -s, --server
           Starts Jetty server.
```

Writing Commands

Most common commands are already available in various standard modules, still often you'd need to write your own. To do that, first create a command class. It should implement `io.bootique.command.Command` interface, though usually it more practical to extend `io.bootique.command.CommandWithMetadata` and provide some metadata used in help and elsewhere:

```
public class MyCommand extends CommandWithMetadata {

    private static CommandMetadata createMetadata() {
        return CommandMetadata.builder(MyCommand.class)
                .description("My command does something important.")
                .build();
    }

    public MyCommand() {
        super(createMetadata());
    }

    @Override
    public CommandOutcome run(Cli cli) {

        // ... run the command here....

        return CommandOutcome.succeeded();
    }
}
```

The command initializes metadata in constructor and implements the "run" method to run its code.

The return CommandOutcome object instructs Bootique what to do when the command finishes. The object contains desired system exit code, and exceptions that occurred during execution. To make the new command available to Bootique, add it to `BQCoreModule’s extender, as was already shown above:

```
public class MyModule implements Module {

    @Override
    public void configure(Binder binder) {
        BQCoreModule.extend(binder).addCommand(MyCommand.class);
    }
}
```

To implement a "daemon" command running forever until it receives an OS signal (e.g. a web server waiting for user requests) , do something like this:

```
@Override
public CommandOutcome run(Cli cli) {

    // ... start some process in a different thread ....

    // now wait till the app is stopped from another thread
    // or the JVM is terminated
    try {
        Thread.currentThread().join();
    } catch (InterruptedException e) {
        // ignore exception or log if needed
    }

    return CommandOutcome.succeeded();
}
```

Injection in Commands

Commands can inject services, just like most other classes in Bootique. There are some specifics though. Since commands are sometimes instantiated, but not executed (e.g. when `--help` is run that lists all commands), it is often desirable to avoid immediate instantiation of all dependencies of a given command. So a common pattern with commands is to inject Guice `Provider` instead of direct dependency:

```
@Inject
private Provider<SomeService> provider;

@Override
public CommandOutcome run(Cli cli) {
    provider.get().someMethod();
}
```

Decorating Commands

Each command typically does a single well-defined thing, such as starting a web server, executing a job, etc. But very often in addition to that main thing you need to do other things. E.g. when a web server is started, you might also want to run a few more commands:

- Before starting the server, run a health check to verify that any external services the app might depend upon are alive.
- Start a job scheduler in the background.
- Start a monitoring "heartbeat" thread.

To run all these "secondary" commands when the main command is invoked, Bootique provides command decorator API. First you create a decorator policy object that specifies one or more secondary commands and their invocation strategy (either *before* the main command, or *in parallel* with it). Second you "decorate" the main command with that policy:

```
CommandDecorator extraCommands = CommandDecorator
  .beforeRun(CustomHealthcheckCommand.class)
  .alsoRun(ScheduleCommand.class)
  .alsoRun(HeartbeatCommand.class);

BQCoreModule.extend(binder).decorateCommand(ServerCommand.class, extraCommands);
```

Based on the specified policy Bootique figures out the sequence of execution and runs the main and the secondary commands.

Chapter 11. Options

Simple Options

In addition to commands, the app can define "options". Options are not associated with any runnable java code, and simply pass command-line values to commands and services. E.g. the standard “--config” option is used by `CliConfigurationSource` service to locate configuration file. Unrecognized options cause application startup errors. To be recognized, options need to be "contributed" to Bootique similar to commands:

```
CliOption option = CliOption
    .builder("email", "An admin email address")
    .valueRequired("email_address").build();

BQCoreModule.extend(binder).addOption(option);
```

To read a value of the option, a service should inject `io.bootique.cli.Cli` object (commands also get this object as a parameter to "run") :

```
@Inject
private Cli cli;

public void doSomething() {
    Collection<String> emails = cli.optionStrings("email");
    // do something with option values....
}
```

Configuration Options

While you can process your own options as described above, options often are just aliases to enable certain pieces of configuration. Bootique supports three flavors of associating options with configuration. Let’s demonstrate them here.

1. Option value sets a config property:

   ```
   // Starting the app with "--my-opt=x" will set "jobs.myjob.param" value to "x"
   BQCoreModule.extend(binder).addOption("jobs.myjob.param", "my-opt");
   ```

2. Option presence sets a property to a predefined value:

   ```
   // Starting the app with "--my-opt" will set "jobs.myjob.param" value to "y"
   BQCoreModule.extend(binder).addOption("jobs.myjob.param", "y", "my-opt");
   ```

3. Option presence loads a config resource, such as a YAML file:

   ```
   // Starting the app with "--my-opt" is equivalent to starting with "--
   config=classpath:xyz.yml"
   BQCoreModule.extend(binder).addConfigFileOption("classpath:xyz.yml", "my-opt");
   ```

The order of config-bound options on the command line is significant, just as the order of “--config” parameters. Bootique merges configuration associated with options from left to right, overriding any preceding configuration if there is an overlap.

Chapter 12. Logging

Loggers in the Code

Standard Bootique modules use SLF4J internally, as it is the most convenient least common denominator framework, and can be easily bridged to other logging implementations. Your apps or modules are not required to use SLF4J, though if they do, it will likely reduce the amount of bridging needed to route all logs to a single destination.

Configurable Logging with Logback

For better control over logging a standard module called `bootique-logback` is available, that integrates Logback framework in the app. It seamlessly bridges SLF4J (so you keep using SLF4J in the code), and allows to configure logging via YAML config file, including appenders (file, console, etc.) and per class/package log levels. Just like any other module, `bootique-logback` can be enabled by simply adding it to the pom.xml dependencies, assuming `autoLoadModules()` is in effect:

```
<dependency>
    <groupId>io.bootique.logback</groupId>
    <artifactId>bootique-logback</artifactId>
</dependency>
```

See `bootique-logback` module documentation for further details.

BootLogger

To perform logging during startup, before DI environment is available and YAML configuration is processed, Bootique uses a special service called `BootLogger`, that is not dependent on SLF4J and is not automatically bridged to Logback. It provides an abstraction for writing to stdout / stderr, as well as conditional "trace" logs sent to stderr. To enable Bootique trace logs, start the app with `-Dbq.trace` as described in the deployment section.

BootLogger is injectable, in case your own code needs to use it. If the default BootLogger behavior is not satisfactory, it can be overridden right in the `main(..)` method, as unlike other services, you may need to change it before DI is available:

```
public class Application {
  public static void main(String[] args) {
     Bootique.app(args).bootLogger(new MyBootLogger()).run();
  }
}
```

Part III. Testing

Chapter 13. Bootique and Testing

Bootique is uniquely suitable to be used as a test framework. Within a single test it allows you to start and stop multiple embedded Bootique runtimes, each with distinct set of modules and distinct YAML configurations, making it a powerful tool for *integration testing*.

Chapter 14. Creating Test Runtimes

Here we'll demonstrate the use of the core test framework. For module-specific test APIs (e.g. `bootique-jdbc-test`), check documentation of those modules or GitHub. To use the core framework, import the following module in the "test" scope:

```
<dependency>
    <groupId>io.bootique</groupId>
    <artifactId>bootique-test</artifactId>
    <scope>test</scope>
</dependency>
```

Then create a `BQTestFactory` in each integration test, annotated with `@Rule` (or `@ClassRule` if you are planning to create a single runtime for all tests in a given class) :

```
public class MyTest {

    @Rule
    public BQTestFactory testFactory = new BQTestFactory();
}
```

Now use the factory to create test runtimes. Each runtime object is essentially an entire Bootique application. It can be used to inspect DI contents, execute a command (including commands that start background processes, such as `--server` and `--schedule`), etc. You don't need to stop the runtime explicitly. `BQTestFactory` will take care of shutdown through JUnit lifecycle.

`testFactory.app()` returns a builder that mimics the API of `Bootique` class, with a few test-related extensions. E.g. it allows to load extra modules, etc.

```
@Test
public void testAbc() {

    BQRuntime runtime = testFactory.app()
        // ensure all classpath modules are included
        .autoLoadModules()
        // add an adhoc module specific to the test
        .module(binder -> binder.bind(MyService.class).to(MyServiceImpl.class))
        .createRuntime();
    // ...
}
```

If you don't need the runtime instance, but rather want to run a command, you'd call `run()` instead of `createRuntime()` (`run()` is an alias for `createRuntime().run()`):

```
@Test
public void testAbc() {

    CommandOutcome result = testFactory.app("--server")
        .autoLoadModules()
        .run();
    // ...
}
```

Chapter 15. Common Test Scenarios

Among the things that can be tested are runtime services with real dependencies, standard output of full Bootique applications (i.e. the stuff that would be printed to the console if this were a real app), network services using real network connections (e.g. your REST API's), and so on. Some examples are given below, outlining common techniques.

Testing Injectable Services

Services can be obtained from test runtime, their methods called, and assertions made about the results of the call:

```
@Test
public void testService() {

    BQRuntime runtime = testFactory.app("--
config=src/test/resources/my.yml").createRuntime();

    MyService service = runtime.getInstance(MyService.class);
    assertEquals("xyz", service.someMethod());
}
```

Testing Network Services

If a test command starts a web server or some other network service, it can be accessed via a URL right after running the server. E.g.:

```
@Test
public void testServer() {

    testFactory.app("--server").run();

    // using JAX-RS client API
    WebTarget base = ClientBuilder.newClient().target("http://localhost:8080/");
    Response r1 = base.path("/somepath").request().get();
    assertEquals(Status.OK.getStatusCode(), r1.getStatus());
    assertEquals("{}", r1.readEntity(String.class));
}
```

Testing Commands

You can emulate a real app execution in a unit test, by running a command and then checking the values of the exist code and `stdin` and `stderr` contents:

```
@Test
public void testCommand() {

    TestIO io = TestIO.noTrace();
    CommandOutcome outcome = testFactory
        .app("--help")
        .bootLogger(io.getBootLogger())
        .run();

    assertEquals(0, outcome.getExitCode());
    assertTrue(io.getStdout().contains("--help"));
    assertTrue(io.getStdout().contains("--config"));
}
```

Testing Module Validity

When you are writing your own modules, you may want to check that they are configured properly for autoloading (i.e. `META-INF/services/io.bootique.BQModuleProvider` is present in the expected place and contains the right provider. There’s a helper class to check for it:

```
@Test
public void testAutoLoadable() {
    BQModuleProviderChecker.testAutoLoadable(MyModuleProvider.class);
}
```

Part IV. Assembly and Deployment

Chapter 16. Runnable Jar

To build a runnable jar, Bootique relies on `maven-shade-plugin`. To simplify its configuration, your app `pom.xml` may inherit from `bootique-parent` pom. In this case configuration would look like this:

```
<parent>
    <groupId>io.bootique.parent</groupId>
    <artifactId>bootique-parent</artifactId>
    <version>0.12</version>
</parent>

...
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-shade-plugin</artifactId>
        </plugin>
    </plugins>
</build>
```

This configuration will build an app with the framework-provided main class, namely `io.bootique.Bootique`. If you want to use a custom main class (and in most cases you do), you will need to redefine Maven `main.class` property:

```
<properties>
    <main.class>com.foo.Application</main.class>
</properties>
```

If you want to avoid inheriting from the framework parent pom, you will need to explicitly provide the following unwieldy configuration similar to the one found in `bootique-parent`:

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-shade-plugin</artifactId>
    <version>2.4.2</version>

    <configuration>
        <createDependencyReducedPom>true</createDependencyReducedPom>
        <filters>
            <filter>
                <artifact>*:*</artifact>
                <excludes>
                    <exclude>META-INF/*.SF</exclude>
                    <exclude>META-INF/*.DSA</exclude>
                    <exclude>META-INF/*.RSA</exclude>
                </excludes>
            </filter>
        </filters>
    </configuration>
    <executions>
        <execution>
            <phase>package</phase>
            <goals>
                <goal>shade</goal>
            </goals>
            <configuration>
                <transformers>
                    <transformer
implementation="org.apache.maven.plugins.shade.resource.ServicesResourceTransformer"
/>
                    <transformer
implementation="org.apache.maven.plugins.shade.resource.ManifestResourceTransformer">
                        <mainClass>${main.class}</mainClass>
                    </transformer>
                </transformers>
            </configuration>
        </execution>
    </executions>
</plugin>
```

Either way, once your pom is configured, you can assemble and run your jar. E.g.:

```
mvn clean package
java -jar target/myapp-1.0.jar
```

Chapter 17. Tracing Bootique Startup

To see what modules are loaded and to trace other events that happen on startup, run your jar with `-Dbq.trace` option. E.g.:

```
java -Dbq.trace -jar target/myapp-1.0.jar --server
```

You may see an output like this:

```
Skipping module 'JerseyModule' provided by 'JerseyModuleProvider' (already provided by
'Bootique')...
Adding module 'BQCoreModule' provided by 'Bootique'...
Adding module 'JerseyModule' provided by 'Bootique'...
Adding module 'JettyModule' provided by 'JettyModuleProvider'...
Adding module 'LogbackModule' provided by 'LogbackModuleProvider'...
```

Kotlin Extensions for Bootique and Bootique Modules

Overview

`bootique-kotlin` contains following modules:

1. Kotlin APIs and extensions for Bootique;
2. Kotlin Script Configuration Module;
3. Configuration and Extensions for Bootique Modules;
4. `JacksonService` which provides `ObjectMapper` with enabled `KotlinModule`.

TL;DR;

- Use `KotlinBootique` instead of `Bootique`;
- Use `KotlinModule` instead of `Module`, you can use `KotlinModule` with `ConfigModule` (just inherit both);
- Use `KotlinBQModuleProvider` instead of `BQModuleProvider`;
- Use extensions defined in Extensions.kt;
- Use `bootique-kotlin-configuration` module to benefit from configuration written in Kotlin.
- Use `bootique-kotlin-jackson` to get `ObjectMapper` with `KotlinModule`.

Getting started

Kotlin **1.2.70** used in project.

Latest stable version: [Maven Central]

Add dependency on needed parts in your `build.gradle`, or `pom.xml`:

```
// Kotlin Extensions for Bootique
compile("io.bootique.kotlin:bootique-kotlin:0.25")

// Kotlin Configuration Module
compile("io.bootique.kotlin:bootique-kotlin-config:0.25")

// Kotlin Configuration Module
compile("io.bootique.kotlin:bootique-kotlin-jackson:0.25")

// Kotlin Configuration and Extensions for Jetty. Also this adds dependency to
bootique-jetty module.
compile("io.bootique.kotlin:bootique-kotlin-jetty:0.25")

// Kotlin Configuration and Extensions for Logback. Also this adds dependency to
bootique-logback module.
compile("io.bootique.kotlin:bootique-kotlin-logback:0.25")

// Kotlin Configuration and Extensions for $moduleName$. Also this adds dependency to
bootique-$moduleName$ module.
compile("io.bootique.kotlin:bootique-kotlin-$moduleName$:0.25")
```

IMPORTANT

`bootique-kotlin` modules doesn't include `kotlin-stdlib-jdk8`, or any other core kotlin libraries, since you can use newer kotlin version and usually you already have `kotlin-stdlib-jdk8` in runtime. So there are list of additional dependencies for different bootique-kotlin modules:

```
bootique-kotlin-config:
  org.jetbrains.kotlin:kotlin-reflect
  org.jetbrains.kotlin:kotlin-script-util
  org.jetbrains.kotlin:kotlin-compiler-embeddable

bootique-kotlin-jackson:
  org.jetbrains.kotlin:kotlin-reflect

bootique-kotlin-logback:
  org.jetbrains.kotlin:kotlin-reflect
```

If you use different kotlin version, it's much simpler to include this libraries with proper version, instead of excluding library version of them and then including again.

If you'd like to use snapshot versions, you have to add bootique snapshot repository:

```
http://maven.objectstyle.org/nexus/content/repositories/bootique-snapshots/
```

For example in gradle it can be done this way:

```
repositories {
    maven { setUrl("http://maven.objectstyle.org/nexus/content/repositories/bootique-
snapshots/") }
}
```

And then use snapshot version: **1.0.RC1-SNAPSHOT**.

Bootique

KotlinBootique

`bootique-kotlin` provides replacement for `Bootique` class - `KotlinBootique`:

```
fun main(args: Array<String>) {
    KotlinBootique(args)
        .module(ApplicationModule::class)
        .exec()
        .exit()
}
```

So no need for extensions for `Bootique` class, `KotlinBootique` provides best experience for developing Bootique apps with Kotlin.

KotlinBQModuleProvider

`KotlinBQModuleProvider` - interface to implement in Bootique Kotlin application instead of `BQModuleProvider`.

```
class ApplicationModuleProvider : KotlinBQModuleProvider {
    override val module = ApplicationModule()
    override val overrides = listOf(BQCoreModule::class)
    override val dependencies = listOf(KotlinConfigModule::class)
}
```

You can see how declarative become module provider.

Extension: `ConfigurationFactory.config`

```
// Using Java Api
configurationFactory.config(SampleFactory::class.java, "sample")

// With Extension
configurationFactory.config(SampleFactory::class, "sample")

// With Extension, reified generics
configurationFactory.config<SampleFactory>("sample")

// Type Inference
@Singleton
@Provides
fun createAppConfiguration(configurationFactory: ConfigurationFactory): SampleFactory
{
    return configurationFactory.config/* No Type Here */(configPrefix)
}
```

Extension: `BQCoreModuleExtender.addCommand`

Straightforward and easy to use extension for contributing commands.

```
BQCoreModule
    .extend(binder)
    .addCommand(ApplicationCommand::class)
```

Extension: `BQCoreModuleExtender.setDefaultCommand`

Also extension for `setDefaultCommand` available.

```
BQCoreModule
    .extend(binder)
    .setDefaultCommand(ApplicationCommand::class)
```

Extensions

See Extensions.kt for sources.

Deprecated Extensions

These extensions deprecated and deleted in 0.25 in favor of `KotlinModule` and `KotlinBootique`.

- `LinkedBindingBuilder.toClass`
- `ScopedBindingBuilder.asSingleton`
- `ScopedBindingBuilder.inScope`

- `Binder.bind`
- `Bootique.module`
- `Bootique.modules`

Guice

KotlinModule

`bootique-kotlin` introduces new module interface to use with kotlin: `KotlinModule`

```
class ApplicationModule : KotlinModule {
    override fun configure(binder: KotlinBinder) {

binder.bind(ShareCountService::class).to(DefaultShareCountService::class).asSingleton(
)
        binder.bind(HttpClient::class).to(DefaultHttpClient::class).asSingleton()
    }
}
```

Extensions

There are few function to help work with `TypeLiteral` and `Key`.

```
// TypeLiteral
typeLiteral<Array<String>>()

// Key
key<List<Callable<A>>>()
```

Configuration Module

Use Kotlin Script for configuration really simple:

1. Create script
2. Override `ConfigurationFactory`

Configuration with Kotlin can be defined in Kotlin Script file:

```
import io.bootique.kotlin.config.modules.config
import io.bootique.kotlin.config.modules.httpConnector
import io.bootique.kotlin.config.modules.jetty

config {
    jetty {
        httpConnector {
            port = 4242
            host = "0.0.0.0"
        }
    }
}
```

Enable Kotlin Script Configuration in Bootique:

With extension:

```
fun main(args: Array<String>) {
    KotlinBootique(args)
        .withKotlinConfig() // Extension function
        .autoLoadModules()
        .exec()
        .exit()
}
```

Using `BQModuleProvider`:

```
fun main(args: Array<String>) {
    KotlinBootique(args)
        .module(KotlinConfigModuleProvider())
        .autoLoadModules()
        .exec()
        .exit()
}
```

You can pass this file as always to bootique:

```
./bin/application --config=classpath:config.kts --server
```

It's even support multiple files (each file contains map of configs):

```
./bin/application --config=classpath:config.kts --config=classpath:config1.kts
--server
```

That’s it! You get autocomplete in IDE, and **code** for configuration!

Bootique Jetty

Define empty config:

```
config {
    jetty {

    }
}
```

Use autocompletion to define configuration.

Use `httpConnector/httpsConnector` extensions to define connectors:

```
jetty {
    httpConnector {
        port = 4242
        host = "192.168.0.1"
        responseHeaderSize = 42
        requestHeaderSize = 13
    }
}
```

Bootique Logback

Define logback configuration:

```
config {
    addConfig("log" to logbackContextFactory(
        logFormat = "[%d{dd/MMM/yyyy:HH:mm:ss}] %t %-5p %c{1}: %m%n",
        useLogbackConfig = false,
        debugLogback = false,
        level = LogbackLevel.warn,
        loggers = mapOf(
            logger(LogbackModuleTest::class, LogbackLevel.error),
            logger("TestLogger", LogbackLevel.trace)
        ),
        appenders = listOf(
            consoleAppender(
                logFormat = "[%d{dd/MMM/yyyy:HH:mm:ss}] %t %-5p %c{1}: %m%n",
                target = ConsoleTarget.stderr
            ),
            fileAppender(logFormat, "abc", timeBasedPolicy(
                fileNamePattern = "Abc_%d",
                totalSize = "2m",
                historySize = 1
            ))
        )
    ))
}
```

Use function for retrieving logger for class:

```
val logger = logger<SomeService>()
```

Or if class is generic:

```
val logger = logger<SomeService<*>>()
```

Bootique Undertow

Define undertow configuration:

```
config {
    addConfig("undertow" to undertowFactory(
        httpListeners = listOf(
            httpListener(1337, "127.0.0.1")
        ),
        workerThreads = 42
    ))
}
```

Bootique Jetty Documentation

Chapter 1. Bootique Integration with Jetty

`bootique-jetty` module embeds Jetty web server in your application. It provides environment for running servlet specification objects (servlets, servlet filters, servlet listeners). Also you will be able to serve static files that are either packaged in the application jar or located somewhere on the filesystem. As things go with Bootique, you will be able to centrally configure both Jetty (e.g. set connector ports) as well as your apps (e.g. map servlet URL patterns and pass servlet parameters).

`bootique-jetty` is "drag-and-drop" just like any other Bootique module. It is enabled by simply adding it to the `pom.xml` dependencies (assuming `autoLoadModules()` is in effect):

```
<dependency>
    <groupId>io.bootique.jetty</groupId>
    <artifactId>bootique-jetty</artifactId>
</dependency>
```

Alternatively you may include an "instrumented" version of `bootique-jetty` that will provide a number of metrics for your running app:

```
<dependency>
    <groupId>io.bootique.jetty</groupId>
    <artifactId>bootique-jetty-instrumented</artifactId>
</dependency>
```

The module provides `--server` command, which starts your web server on foreground:

```
java -jar my.jar --server
...
i.b.j.s.ServerFactory - Adding listener i.b.j.s.DefaultServletEnvironment
i.b.j.s.h.ContextHandler - Started
o.e.j.s.ServletContextHandler@1e78c66e{/myapp,null,AVAILABLE}
i.b.j.s.ServerConnector - Started
ServerConnector@41ccbaa{HTTP/1.1,[http/1.1]}{0.0.0.0:8080}
i.b.j.s.Server - Started @490ms
```

Various aspects of the Jetty container, such as listen port, thread pool size, etc. can be configured in a normal Bootique way via YAML, as detailed below in the "Configuration Reference" chapter.

Chapter 2. Programming Jetty Applications

You can write servlet specification objects (servlets, filters, listeners) as you'd do it under JavaEE, except that there's no `.war` and no `web.xml`. There's only your application, and you need to let

Bootique know about your objects and how they should be mapped to request URLs. Let’s start with servlets.

Servlets

The easiest way to add a servlet to a Bootique app is to annotate it with `@WebServlet`, providing name and url patterns:

```
@WebServlet(
    name = "myservlet",
    urlPatterns = "/b",
    initParams = {
        @WebInitParam(name = "p1", value = "v1"),
        @WebInitParam(name = "p2", value = "v2")
    }
)
public class AnnotatedServlet extends HttpServlet {
    //...
}
```

The "name" annotation is kind of important as it would allow to override annotation values in the YAML, as described in the "Configuration Reference" chapter. A servlet created this way can inject any services it might need using normal Guice injection.

Next step is adding it to Bootique via `JettyModule` contribution API called from your application Module:

```
@Override
public void configure(Binder binder) {
    JettyModule.extend(binder).addServlet(AnnotatedServlet.class);
}
```

But what if you are deploying a third-party servlet that is not annotated? Or annotation values are off in the context of your application? There are two choices. The first is to subclass such servlets and annotate the subclasses that you control.

The second is to wrap your servlet in a special metadata object called `MappedServlet`, providing all the mapping information in that wrapper. This is a bit too verbose, but can be a good way to define the mapping that is not otherwise available:

```
@Override
public void configure(Binder binder) {
    MappedServlet mappedServlet = new MappedServlet(
       new MyServlet(),
       Collections.singleton("/c"),
       "myservlet");

    JettyModule.extend(binder).addMappedServlet(mappedServlet);
}
```

Finally if we need to use MappedServlet for mapping servlet URLs and parameters, but also need the ability to initialize the underlying servlet with environment dependencies, we can do something like this:

```
@Singleton
@Provides
MappedServlet<MyServlet> provideMyServlet(MyService1 s1, MyService2 s2) {
    MyServlet servlet = new MyServlet(s1, s2);
    return new MappedServlet<>(servlet, Collections.singleton("/c"), "myservlet");
}

// must use TypeLiteral to contribute MappedServlet<MyServlet> to the servlet
collection
@Override
public void configure(Binder binder) {
    TypeLiteral<MappedServlet<Servlet2>> tl = new
TypeLiteral<MappedServlet<MyServlet>>() {};
    JettyModule.extend(binder).addMappedServlet(tl);
}
```

Servlet Filters

Just like servlets, you can annotate and register your filters:

```
@WebFilter(
    filterName = "f1",
    urlPatterns = "/b/*",
    initParams = {
        @WebInitParam(name = "p1", value = "v1"),
        @WebInitParam(name = "p2", value = "v2")
    }
)
public class AnnotatedFilter implements Filter { .. }
```

```
@Override
public void configure(Binder binder) {
    JettyModule.extend(binder).addFilter(AnnotatedFilter.class);
}
```

And just like with servlets you can use `MappedFilter` and `JettyModule.extend(..).addMappedFilter` to wrap a filter in app-specific metadata.

Listeners

Listeners are simpler then servlets or filters. All you need is to create classes that implement one of servlet specification listener interfaces (`ServletContextListener`, `HttpSessionListener`, etc.) and bind them in your app:

```
@Override
public void configure(Binder binder) {
    JettyModule.extend(binder).addListener(MyListener.class);
}
```

Listeners can rely on injection to obtain dependencies, just like servlets and filters.

Serving Static Files

TODO

Chapter 3. Configuration Reference

jetty

```
jetty:
  context: "/myapp"
  maxThreads: 100
  params:
    a: a1
    b: b2
```

"jetty" is a root element of the Jetty configuration and is bound to a `ServerFactory` object. It supports the following properties:

Table 1. "jetty" Element Property Reference

Property	Default	Description
`compression`	`true`	A boolean specifying whether gzip compression should be supported. When enabled (default), responses will be compressed if a client indicates it supports compression via `"Accept-Encoding: gzip"` header.
`connector`	*N/A (deprecated since 0.18)*	*Deprecated as more than one connector is supported. Use* \`connectors\` *instead.* An object specifying properties of the web connector.
`connectors`	a single HTTP connector on port 8080	A list of connectors. Each connector listens on a single port. There can be HTTP or HTTPS connectors. See `jetty.connectors` below.
`context`	`/`	Web application context path.
`idleThreadTimeout`	`60000`	A period in milliseconds specifying how long until an idle thread is terminated.
`filters`	empty map	A map of servlet filter configurations by filter name. See `jetty.filters` below.
`maxThreads`	`1024`	Maximum number of request processing threads in the pool.
`minThreads`	`4`	Initial number of request processing threads in the pool.
`maxQueuedRequests`	`1024`	Maximum number of requests to queue if the thread pool is busy.
`params`	empty map	A map of arbitrary key/value parameters that are used as "init" parameters of the ServletContext.
`servlets`	empty map	A map of servlet configurations by servlet name. See `jetty.servlets` below.
`sessions`	`true`	A boolean specifying whether servlet sessions should be supported by Jetty.

Property	Default	Description
`staticResourceBase`	none	Defines a base location for resources of the Jetty context. It can be a filesystem path, a URL or a special “classpath:” URL (which gives the ability to bundle resources in the app, not unlike a JavaEE .war file). + For security reasons this annotation has to be set explicitly. There’s no default. + This setting only makes sense when some form of "default" servlet is in use, that will be responsible for serving static resources. See `JettyModule.contributeStaticServlet(..)` or `JettyModule.contributeDefaultServlet(..)`. Such servlet will use the path defined here, unless overridden via servlet parameters. For the list fo servlet parameters see Jetty default servlet docs.
`compactPath`	`false`	True if URLs are compacted to replace multiple '/'s with a single '/'

jetty.connectors

```
jetty:
  connectors:
    - port: 9999
    - port: 9998
      type: https
```

"jetty.connectors" element configures one or more web connectors. Each connector listens on a specified port and has an associated protocol (http or https). If no connectors are provided, Bootique will create a single HTTP connector on port 8080.

HTTPS connectors require an SSL certificate (real or self-signed), stored in a keystore. Jetty documentaion on the subject should help with generating a certificate. In its simplest form it may look like this:

```
keytool -keystore src/main/resources/mykeystore \
       -alias mycert -genkey -keyalg RSA -sigalg SHA256withRSA -validity 365
```

Table 2. HTTP connector property reference

Property	Default	Description
`type`	N/A	Connector type. To use HTTP connector, this value must be set to "http", or omitted all together ("http" is the default).
`port`	`8080`	A port the connector listens on.
`requestHeaderSize`	`8192`	A max size in bytes of Jetty request headers and GET URLs.
`responseHeaderSize`	`8192`	A max size in bytes of Jetty response headers.

Table 3. HTTPS connector property reference

Property	Default	Description
`type`	N/A	Connector type. To use HTTPS connector, this value must be set to "https".
`port`	`8080`	A port the connector listens on.
`requestHeaderSize`	`8192`	A max size in bytes of Jetty request headers and GET URLs.
`responseHeaderSize`	`8192`	A max size in bytes of Jetty response headers.
`keyStore`		Required. A resource pointing to the keystore that has server SSL certificate. Can be a "classpath:" resource, etc.
`keyStorePassword`	`changeit`	A password to access the keystore.
`certificateAlias`		An optional name of the certificate in the keystore, if there’s more than one certificate.

jetty.filters

```
jetty:
  filters:
    f1:
      urlPatterns:
        - '/a/*/'
        - '/b/*'
      params:
        p1: v1
        p2: v2
    f2:
      params:
        p3: v3
        p4: v4
```

TODO

jetty.servlets

```
jetty:
  servlets:
    s1:
      urlPatterns:
        - '/myservlet'
        - '/someotherpath'
      params:
        p1: v1
        p2: v2
    s2:
      params:
        p3: v3
        p4: v4
    default:
      params:
        resourceBase: /var/www/html
```

TODO

Bootique Logback Documentation

Bootique Integration with Logback

As mentioned in Bootique general documentation on logging, standard modules rely on SLF4J loggers that can be easily bridged to various advanced logging frameworks. Same maximally neutral logging approach is reasonable to apply in the user modules as well.

`bootique-logback` is a "drag-and-drop" module integrating Bootique logging with Logback logging framework. Just like any other module, `bootique-logback` can be enabled by simply adding it to the `pom.xml` dependencies, assuming `autoLoadModules()` is in effect:

```
<dependency>
    <groupId>io.bootique.logback</groupId>
    <artifactId>bootique-logback</artifactId>
</dependency>
```

Without further configuration it would log everything to console using INFO level. Configuration can be provided via YAML, as shown in the following section. Configuration options include per class and package log levels configuration, a choice of appenders, etc.

Configuration Reference

log

```
log:
  level: warn
  loggers:
    com.foo:
      level: debug
    com.example:
      level: debug
  appenders:
    - type: file
      logFormat: '%c{20}: %m%n'
      file: target/logback/testRun_Debug.log
```

"log" is a root element of the Logback configuration and is bound to a `LogbackContextFactory` object. It supports the following properties:

Table 4. "log" Element Property Reference

Property	Default	Description
`appenders`	console appender	A list of appenders to output logs to. See below.
`level`	`INFO`	A default logging level. Can be overridden per logger.
`loggers`	none	A map of logger factories by logger name. Logger name is a package name (applied recursively) or class name. See below.
`useLogbackConfig`	`false`	If true, all other logback settings areignored and the user is expected to provide its own config file per Logback documentation. This is only needed for a few advanced options not directly available via Bootique config.

log.appenders

```
log:
  appenders:
    - type: file
      logFormat: '%c{20}: %m%n'
      file: /var/log/myapp.log
    - type: console
      logFormat: '%c{20}: %m%n'
```

Lists appenders to sends the logs to. If the list is empty, console appender is used with default settings. Currently available appenders are "console" and "file":

Table 5. "console" Appender Property Reference

Property	Default	Description
`logFormat`	`%-5p [%d{ISO8601,UTC}] %thread %c{20}: %m%n%rEx`	A format of the log lines. See Logback PatternLayout for details.
`target`	`stdout`	Whether to log to standard output or standard error. Possible values are `stdout` and `stderr`

Table 6. "file" Appender Property Reference

Property	Default	Description
logFormat	%-5p [%d{ISO8601,UTC}] %thread %c{20}: %m%n%rEx	A format of the log lines. See Logback PatternLayout for details.
file	none	A name of a file to send the log to.
rollingPolicy	none	An object that defines a log rotation policy. Examples are given below.

There are a few ways log file rotation can be configured for the "file" appender, as defined by the rollingPolicy. Out of the box the following Logback policies are supported: fixedWindow, time, sizeAndTime.

"fixedWindow" Rolling Policy

```
log:
  appenders:
    - type: file
      logFormat: '%c{20}: %m%n'
      file: /var/log/myapp.log
      rollingPolicy:
        type: fixedWindow
        fileNamePattern: '/var/log/myapp-%i.log'
        historySize: 5
        fileSize: 20
```

"fixedWindow" policy rotates the main log file when it reaches a certain size, keeping one or more rotated files.

Table 7. "fixedWindow" rolling policy Property Reference

Property	Default	Description
fileNamePattern	none	A pattern of rotated file name. Must contain %i somewhere in the pattern (replaced by a number during rotation).
historySize	none (unlimited)	A max number of rotated files to keep.
fileSize	none	Max file size that causes rotation. Expressed in bytes, kilobytes, megabytes or gigabytes by suffixing a numeric value with KB, MB and respectively GB. For example: 5000000, 5000KB, 5MB and 2GB.

"time" Rolling Policy

```
log:
  appenders:
    - type: file
      logFormat: '%c{20}: %m%n'
      file: /var/log/myapp.log
      rollingPolicy:
        type: time
        fileNamePattern: '/var/log/myapp-%d{yyyyMMddHHmmss}.log'
```

"time" policy rotates the main log file at a fixed time interval determined by the file name pattern.

Table 8. "time" rolling policy Property Reference

Property	**Default**	**Description**
`fileNamePattern`	none	A pattern of rotated file name. Its value should consist of the name of the file, plus a suitably placed %d conversion specifier. The %d conversion specifier may contain a date-and-time pattern as specified by the `java.text.SimpleDateFormat` class. If the date-and-time pattern is omitted, then the default pattern `yyyy-MM-dd` is assumed. The rollover interval is inferred from the value of the pattern.
`historySize`	none (unlimited)	A max number of rotated files to keep.
`totalSize`	none	Max size of all log files combined. Expressed in bytes, kilobytes, megabytes or gigabytes by suffixing a numeric value with KB, MB and respectively GB. For example: 5000000, 5000KB, 5MB and 2GB.

"sizeAndTime" Rolling Policy

```
log:
  appenders:
    - type: file
      logFormat: '%c{20}: %m%n'
      file: /var/log/myapp.log
      rollingPolicy:
        type: sizeAndTime
        fileNamePattern: '/var/log/myapp-%d{yyyyMMddHHmmss}.%i.log'
        historySize: 5
        fileSize: 50
        totalSize: 150
```

"sizeAndTime" policy rotates the main log file either at a fixed time interval determined by the file name pattern or when the log file reaches a certain size.

Table 9. "sizeAndTime" rolling policy Property Reference

Property	**Default**	**Description**
`fileNamePattern`	none	A pattern of rotated file name. Its value should consist of the name of the file, plus a suitably placed %d conversion specifier. The %d conversion specifier may contain a date-and-time pattern as specified by the `java.text.SimpleDateFormat` class. If the date-and-time pattern is omitted, then the default pattern `yyyy-MM-dd` is assumed. The rollover interval is inferred from the value of the pattern.
`historySize`	none (unlimited)	A max number of rotated files to keep.
`totalSize`	none	Max size of all log files combined. Expressed in bytes, kilobytes, megabytes or gigabytes by suffixing a numeric value with KB, MB and respectively GB. For example: 5000000, 5000KB, 5MB and 2GB.
`fileSize`	none	Max file size that causes rotation. Expressed in bytes, kilobytes, megabytes or gigabytes by suffixing a numeric value with KB, MB and respectively GB. For example: 5000000, 5000KB, 5MB and 2GB.

log.loggers

```
log:
  loggers:
    com.foo:
      level: debug
    com.example:
      level: debug
```

This is a map of logger factories by logger name. Logger name is either a package name (applied recursively to subpackages and their classes) or a class name. Each LoggerFactory has the following properties:

Table 10. Logger Property Reference

Property	Default	Description
`level`	`INFO`	Log level for a particular logger. Can be OFF, ERROR, WARN, INFO, DEBUG, TRACE, ALL

Logback Sentry Module

Provides Sentry integration with Bootique-Logback.

Setup

Get DSN from sentry.io or from your own instance of Sentry.

Add bootique-logback-sentry to your build tool:

Maven

```
<dependency>
    <groupId>io.bootique.logback</groupId>
    <artifactId>bootique-logback-sentry</artifactId>
</dependency>
```

Gradle

```
compile("io.bootique.logback:bootique-logback-sentry:$bootiqueVersion")
```

NOTE **bootique-logback-sentry** is a part of bootique-bom, and version can be imported from there.

Write Configuration

config.yml:

```
log:
  level: warn
  appenders:
    - type: console
      logFormat: '[%d{dd/MMM/yyyy:HH:mm:ss}] %t %-5p %c{1}: %m%n'
    - type: sentry
      dsn: 'your_dsn_here'
      serverName: aurora
      environment: development
      release: 42.0
      extra:
        extra1: value1
        extra2: value2
        extra3: value3
      minLevel: error
      distribution: x86
      applicationPackages:
        - "io.bootique.logback"
        - "com.myapp.package"
      commonFramesEnabled: true
      tags:
        tag1: value1
        tag2: value2
```

`SentryClientFactory` can be provided by overriding `SentryClientFactory` bean from `LogbackSentryModule`.

Also DSN can be provided via environment variable SENTRY_DSN.