```
import com.google.inject.*;
import org.reflections.Reflections;
import java.lang.annotation.*;
import java.lang.reflect.Field;
import java.lang.reflect.Method;
import java.util.Set;
import java.util.HashMap;
import java.util.Map;

// @Value Annotation for Injecting Config Values
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.FIELD)
@interface Value {
    String value();
}

// @AutoBind Annotation for Automatic Binding
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
@interface AutoBind {}

// @Controller Annotation for Marking Controllers
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
@interface Controller {
    String value();
}

// @RequestMapping for Method-Level Routing
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
@interface RequestMapping {
    String value();
}

// Config Provider for Injecting @Value Annotations
class ConfigProvider implements Provider<String> {
    private final ConfigLoader configLoader;
    private final String key;

    @Inject
    public ConfigProvider(ConfigLoader configLoader, Field field) {
        this.configLoader = configLoader;
        this.key = field.getAnnotation(Value.class).value();
    }

    @Override
    public String get() {
        Object value = configLoader.getProperty(key);
        if (value == null) {
            throw new ProvisionException("Missing config key: " + key);
        }
        return value.toString();
    }
}

// Config Loader Interface
interface ConfigLoader {
    Object getProperty(String key);
}

// Auto-Scanning Module for Guice
class AutoScanModule extends AbstractModule {
    private final String basePackage;
```

```
    public AutoScanModule(String basePackage) {
        this.basePackage = basePackage;
    }

    @Override
    protected void configure() {
        Reflections reflections = new Reflections(basePackage);

        // Auto-bind all classes annotated with @AutoBind
        Set<Class<?>> autoBindClasses = reflections.getTypesAnnotatedWith(AutoBind.class);
        for (Class<?> clazz : autoBindClasses) {
            bind(clazz);
        }

        // Auto-bind all classes annotated with @Controller
        Set<Class<?>> controllerClasses = reflections.getTypesAnnotatedWith(Controller.class);
        for (Class<?> clazz : controllerClasses) {
            bind(clazz).in(Scopes.SINGLETON);
        }
    }
}

// Example Usage
@AutoBind
class MyService {
    void serve() {
        System.out.println("Service is running...");
    }
}

@Controller("/api")
class MyController {
    private final MyService service;

    @Inject
    public MyController(MyService service) {
        this.service = service;
    }

    @RequestMapping("/hello")
    public void sayHello() {
        System.out.println("Hello from MyController");
    }

    @RequestMapping("/serve")
    public void handleRequest() {
        service.serve();
    }
}

// Example Usage of @Value
@AutoBind
class ConfiguredService {
    @Inject
    private ConfigLoader configLoader;

    @Value("app.name")
    private String appName;

    public void printConfig() {
        System.out.println("Application Name: " + appName);
    }
}

// Simple Dependency Injection Framework
```

```
// Simple Router to Call Methods Based on Request Mapping
class Router {
    private final Map<String, Method> routes = new HashMap<>();
    private final Injector injector;

    public Router(Injector injector) {
        this.injector = injector;
        registerRoutes();
    }

    private void registerRoutes() {
        Reflections reflections = new Reflections("com.example");
        Set<Class<?>> controllers = reflections.getTypesAnnotatedWith(Controller.class);
        for (Class<?> controller : controllers) {
            for (Method method : controller.getDeclaredMethods()) {
                if (method.isAnnotationPresent(RequestMapping.class)) {
                    String path = method.getAnnotation(RequestMapping.class).value();
                    routes.put(path, method);
                }
            }
        }
    }

    public void handleRequest(String path) {
        Method method = routes.get(path);
        if (method != null) {
            try {
                Object instance = injector.getInstance(method.getDeclaringClass());
                method.invoke(instance);
            } catch (Exception e) {
                e.printStackTrace();
            }
        } else {
            System.out.println("No route found for: " + path);
        }
    }
}

// Main Application Setup
public class Main {
    public static void main(String[] args) {
        Injector injector = Guice.createInjector(new AutoScanModule("com.example"));
        Router router = new Router(injector);

        // Simulating Requests
        router.handleRequest("/api/hello");
        router.handleRequest("/api/serve");

        ConfiguredService configuredService = injector.getInstance(ConfiguredService.class);
        configuredService.printConfig();
    }
}
```

```
------------------------------


import com.google.inject.*;
import org.reflections.Reflections;
import java.lang.annotation.*;
import java.lang.reflect.Field;
import java.lang.reflect.Method;
import java.util.Set;
import java.util.HashMap;
import java.util.Map;
import org.h2.jdbcx.JdbcDataSource;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.apache.catalina.LifecycleException;
import org.apache.catalina.startup.Tomcat;

// @Value Annotation for Injecting Config Values
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.FIELD)
@interface Value {
    String value();
}

// @AutoBind Annotation for Automatic Binding
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
@interface AutoBind {}

// @Controller Annotation for Marking Controllers
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
@interface Controller {
    String value();
}

// @RequestMapping for Method-Level Routing
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
@interface RequestMapping {
    String value();
}

// Config Provider for Injecting @Value Annotations
class ConfigProvider implements Provider<String> {
    private final ConfigLoader configLoader;
    private final String key;
```

```
    @Inject
    public ConfigProvider(ConfigLoader configLoader, Field field) {
        this.configLoader = configLoader;
        this.key = field.getAnnotation(Value.class).value();
    }

    @Override
    public String get() {
        Object value = configLoader.getProperty(key);
        if (value == null) {
            throw new ProvisionException("Missing config key: " + key);
        }
        return value.toString();
    }
}

// Config Loader Interface
interface ConfigLoader {
    Object getProperty(String key);
}

// Auto-Scanning Module for Guice
class AutoScanModule extends AbstractModule {
    private final String basePackage;

    public AutoScanModule(String basePackage) {
        this.basePackage = basePackage;
    }

    @Override
    protected void configure() {
        Reflections reflections = new Reflections(basePackage);

        // Auto-bind all classes annotated with @AutoBind
        Set<Class<?>> autoBindClasses = reflections.getTypesAnnotatedWith(AutoBind.class);
        for (Class<?> clazz : autoBindClasses) {
            bind(clazz);
        }

        // Auto-bind all classes annotated with @Controller
        Set<Class<?>> controllerClasses = reflections.getTypesAnnotatedWith(Controller.class);
        for (Class<?> clazz : controllerClasses) {
            bind(clazz).in(Scopes.SINGLETON);
        }
    }
}

// H2 Database Connection
class Database {
    private static final String DB_URL = "jdbc:h2:mem:test;DB_CLOSE_DELAY=-1";
    private static Connection connection;

    static {
        try {
            JdbcDataSource ds = new JdbcDataSource();
            ds.setURL(DB_URL);
            connection = ds.getConnection();
            connection.createStatement().execute("CREATE TABLE users (username VARCHAR(255), password VARCHAR(255))");
        } catch (SQLException e) {
            throw new RuntimeException("Failed to initialize database", e);
        }
    }

    public static void insertUser(String username, String password) {
        try (PreparedStatement stmt = connection.prepareStatement("INSERT INTO users (username, password) VALUES (?, ?)")) {
```

```
        try (PreparedStatement stmt = connection.prepareStatement("INSERT INTO users (username, password) VALUES (?, ?)")) {
            stmt.setString(1, username);
            stmt.setString(2, password);
            stmt.executeUpdate();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }

    public static boolean fetchUser(String username, String password) {
        try (PreparedStatement stmt = connection.prepareStatement("SELECT * FROM users WHERE username = ? AND password = ?")) {
            stmt.setString(1, username);
            stmt.setString(2, password);
            ResultSet rs = stmt.executeQuery();
            return rs.next();
        } catch (SQLException e) {
            e.printStackTrace();
        }
        return false;
    }
}

// Router for handling request mappings
class Router {
    private final Map<String, Method> routeMap = new HashMap<>();
    private final Map<Class<?>, Object> controllerInstances = new HashMap<>();

    public Router(Injector injector, String basePackage) {
        Reflections reflections = new Reflections(basePackage);
        Set<Class<?>> controllerClasses = reflections.getTypesAnnotatedWith(Controller.class);

        for (Class<?> controllerClass : controllerClasses) {
            Object controllerInstance = injector.getInstance(controllerClass);
            controllerInstances.put(controllerClass, controllerInstance);
            for (Method method : controllerClass.getDeclaredMethods()) {
                if (method.isAnnotationPresent(RequestMapping.class)) {
                    String path = method.getAnnotation(RequestMapping.class).value();
                    routeMap.put(path, method);
                }
            }
        }
    }

    public void handleRequest(String path) {
        Method method = routeMap.get(path);
        if (method != null) {
            Object controllerInstance = controllerInstances.get(method.getDeclaringClass());
            try {
                method.invoke(controllerInstance);
            } catch (IllegalAccessException | InvocationTargetException e) {
                e.printStackTrace();
            }
        } else {
            System.out.println("No handler found for: " + path);
        }
    }
}

// Embedded Tomcat Server
class EmbeddedTomcat {
    public static void start(Router router) throws LifecycleException {
        Tomcat tomcat = new Tomcat();
        tomcat.setPort(8080);
        tomcat.addContext("", null);
        tomcat.addServlet("", "dispatcher", new HttpServlet() {
            @Override
```

```
            protected void service(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
                String path = req.getPathInfo();
                router.handleRequest(path);
                resp.getWriter().write("Handled request: " + path);
            }
        }).addMapping("/*");
        tomcat.start();
        tomcat.getServer().await();
    }
}

// Main Application Setup
public class Main {
    public static void main(String[] args) throws LifecycleException {
        String basePackage = "com.example";
        Injector injector = Guice.createInjector(new AutoScanModule(basePackage));
        Router router = new Router(injector, basePackage);
        EmbeddedTomcat.start(router);
    }
}
```

```
import com.google.inject.*;
import org.reflections.Reflections;
import java.lang.annotation.*;
import java.lang.reflect.*;
import java.util.*;
import org.h2.jdbcx.JdbcDataSource;
import java.sql.*;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import org.apache.catalina.LifecycleException;
import org.apache.catalina.startup.Tomcat;

// @Value Annotation for Injecting Config Values
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.FIELD)
@interface Value {
    String value();
}

// @AutoBind Annotation for Automatic Binding
@Retention(RetentionPolicy.RUNTIME)
```

```
@Target(ElementType.TYPE)
@interface AutoBind {}

// @Controller Annotation for Marking Controllers
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
@interface Controller {
    String value();
}

// @RequestMapping for Method-Level Routing
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
@interface RequestMapping {
    String value();
}

// APIRequest class for request metadata
class APIRequest {
    private final String path;
    private final String contextPath;
    private final Map<String, String> headers;
    private final Map<String, String> queryParams;
    private final Map<String, String> cookies;
    private final String body;

    public APIRequest(HttpServletRequest request) throws IOException {
        this.path = request.getPathInfo();
        this.contextPath = request.getContextPath();
        this.headers = extractHeaders(request);
        this.queryParams = extractQueryParams(request);
        this.cookies = extractCookies(request);
        this.body = extractBody(request);
    }

    private Map<String, String> extractHeaders(HttpServletRequest request) {
        Map<String, String> headers = new HashMap<>();
        Enumeration<String> headerNames = request.getHeaderNames();
        while (headerNames.hasMoreElements()) {
            String name = headerNames.nextElement();
            headers.put(name, request.getHeader(name));
        }
        return headers;
    }

    private Map<String, String> extractQueryParams(HttpServletRequest request) {
        Map<String, String> queryParams = new HashMap<>();
        request.getParameterMap().forEach((key, value) -> queryParams.put(key, value[0]));
        return queryParams;
    }

    private Map<String, String> extractCookies(HttpServletRequest request) {
        Map<String, String> cookies = new HashMap<>();
        if (request.getCookies() != null) {
            for (Cookie cookie : request.getCookies()) {
                cookies.put(cookie.getName(), cookie.getValue());
            }
        }
        return cookies;
    }

    private String extractBody(HttpServletRequest request) throws IOException {
        StringBuilder body = new StringBuilder();
        BufferedReader reader = request.getReader();
        String line;
        while ((line = reader.readLine()) != null) {
```

```
            body.append(line).append("\n");
        }
        return body.toString().trim();
    }

    public String getPath() { return path; }
    public String getContextPath() { return contextPath; }
    public Map<String, String> getHeaders() { return headers; }
    public Map<String, String> getQueryParams() { return queryParams; }
    public Map<String, String> getCookies() { return cookies; }
    public String getBody() { return body; }
}

// Config Provider for Injecting @Value Annotations
class ConfigProvider implements Provider<String> {
    private final ConfigLoader configLoader;
    private final String key;

    @Inject
    public ConfigProvider(ConfigLoader configLoader, Field field) {
        this.configLoader = configLoader;
        this.key = field.getAnnotation(Value.class).value();
    }

    @Override
    public String get() {
        Object value = configLoader.getProperty(key);
        if (value == null) {
            throw new ProvisionException("Missing config key: " + key);
        }
        return value.toString();
    }
}

// Config Loader Interface
interface ConfigLoader {
    Object getProperty(String key);
}

// Auto-Scanning Module for Guice
class AutoScanModule extends AbstractModule {
    private final String basePackage;

    public AutoScanModule(String basePackage) {
        this.basePackage = basePackage;
    }
 @Override
    protected void configure() {
        Reflections reflections = new Reflections(basePackage);
   // Auto-bind all classes annotated with @AutoBind
        Set<Class<?>> autoBindClasses = reflections.getTypesAnnotatedWith(AutoBind.class);
        for (Class<?> clazz : autoBindClasses) {
            bind(clazz);
        }
        Set<Class<?>> controllerClasses = reflections.getTypesAnnotatedWith(Controller.class);
        for (Class<?> clazz : controllerClasses) {
            bind(clazz).in(Scopes.SINGLETON);
        }
    }
}

// Router for handling request mappings
class Router {
    private final Map<String, Method> routeMap = new HashMap<>();
    private final Map<Class<?>, Object> controllerInstances = new HashMap<>();
```

```
    public Router(Injector injector, String basePackage) {
        Reflections reflections = new Reflections(basePackage);
        Set<Class<?>> controllerClasses = reflections.getTypesAnnotatedWith(Controller.class);

        for (Class<?> controllerClass : controllerClasses) {
            Object controllerInstance = injector.getInstance(controllerClass);
            controllerInstances.put(controllerClass, controllerInstance);
            for (Method method : controllerClass.getDeclaredMethods()) {
                if (method.isAnnotationPresent(RequestMapping.class)) {
                    String path = method.getAnnotation(RequestMapping.class).value();
                    routeMap.put(path, method);
                }
            }
        }
    }

    public void handleRequest(HttpServletRequest request, HttpServletResponse response) {
        try {
            APIRequest apiRequest = new APIRequest(request);
            Method method = routeMap.get(apiRequest.getPath());
            if (method != null) {
                Object controllerInstance = controllerInstances.get(method.getDeclaringClass());
                method.invoke(controllerInstance, apiRequest);
            } else {
                response.sendError(HttpServletResponse.SC_NOT_FOUND, "No handler found for: " + apiRequest.getPath());
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
// H2 Database Connection
class Database {
    private static final String DB_URL = "jdbc:h2:mem:test;DB_CLOSE_DELAY=-1";
    private static Connection connection;

    static {
        try {
            JdbcDataSource ds = new JdbcDataSource();
            ds.setURL(DB_URL);
            connection = ds.getConnection();
            connection.createStatement().execute("CREATE TABLE users (username VARCHAR(255), password VARCHAR(255))");
        } catch (SQLException e) {
            throw new RuntimeException("Failed to initialize database", e);
        }
    }

    public static void insertUser(String username, String password) {
        try (PreparedStatement stmt = connection.prepareStatement("INSERT INTO users (username, password) VALUES (?, ?)")) {
            stmt.setString(1, username);
            stmt.setString(2, password);
            stmt.executeUpdate();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }

    public static boolean fetchUser(String username, String password) {
        try (PreparedStatement stmt = connection.prepareStatement("SELECT * FROM users WHERE username = ? AND password = ?")) {
            stmt.setString(1, username);
            stmt.setString(2, password);
            ResultSet rs = stmt.executeQuery();
            return rs.next();
        } catch (SQLException e) {
            e.printStackTrace();
        }
```

```
        }
        return false;
    }
}

}

// Embedded Tomcat Server
class EmbeddedTomcat {
    public static void start(Router router) throws LifecycleException {
        Tomcat tomcat = new Tomcat();
        tomcat.setPort(8080);
        tomcat.addContext("", null);
        tomcat.addServlet("", "dispatcher", new HttpServlet() {
            @Override
            protected void service(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
                router.handleRequest(req, resp);
            }
        }).addMapping("/*");
        tomcat.start();
        tomcat.getServer().await();
    }
}

// Main Application Setup
public class Main {
    public static void main(String[] args) throws LifecycleException {
        String basePackage = "com.example";
        Injector injector = Guice.createInjector(new AutoScanModule(basePackage));
        Router router = new Router(injector, basePackage);
        EmbeddedTomcat.start(router);
    }
}
```